


code.sd

أول إصدار:  ذي القعدة 1433 هجرية الموافق أكتوبر 2012 ميلادية

الإصدار الحالي: ربيع ثاني 1437 هجرية الموافق 30 يناير 2016 ميلادية

# مقدمة

بسم الله الرحمن الرحيم والصلاة والسلام على أشرف الأنبياء والمرسلين، نبينا محمد وعلى آله وصحبه أجمعين.
أما بعد. الهدف من هذا الكُتيب تعريف المبرمج في فترة وجيزة وكمدخل سريع للغة البرمجة جافا باستخدام أداة التطوير NetBeans. وبهذا يكون هذا الكُتيب موجه فقط لمن لديه خبرة في لغة برمجة أخرى حتى لو كانت قليلة. كذلك يُمكن الإستفادة من هذا الكتاب كمقدمة لتعلم برمجة الموبايل بإستخدام جافا، مثل نظام أندرويد أو جافا موبايل وذلك لأن أساس اللغة واحدة.

# لغة جافا

لغة جافا ظهرت في عام 1995 (أي قبل أكثر من عشرين عاماً) وهي لغة متعددة الأغراض ومتعددة المنصات تصلح لعدد كبير من التطبيقات. ومترجم جافا يقوم بإنتاج ملفات في شكل Byte code وهو يختلف عن الملفات التفيذية التي تنتج عن لغات البرمجة الأخرى مثل سي وباسكال. وتحتاج البرامج المكتوبة بلغة جافا إلى منصة في أنظمة التشغيل المختلفة لتتمكن برامجها من العمل في هذه الأنظمة. وهذه المنصة تُسمة آلة جافا الإفتراضية Java Virtual Machine أو إختصاراً بـ JVM أو Java Run-time.

# آلة جافا الإفتراضية JVM

تتوفر هذه المنصة في عدد كبير من أنظمة التشغيل، ولابد  من التأكد من وجود هذه المنصة أو الآلة الإفتراضية قبل تشغيل برنامج جافا. وكل نظام تشغيل يحتاج لآلة إفتراضية خاصة به. مثلاً نظام وندوز 32 بت يحتاج لآلة إفتراضية مخصصة لوندوز 32 بت، ووندوز 64 بت يحتاج لآلة إفتراضية 64 بت. وهذا مثال لإسم ملف لتثبيت آلة جافا الإفتراضية لنظام وندوز 64 بت:

```
jre-8u51-windows-x64.exe
```

وهو يُمثل نسخة جافا 1.8 أو مايُسمى جافا 8 وهي آخر نسخة من الجافا متوفرة لحظة إعادة تحرير هذا الكتاب.

واسم الملف التالي يُمثل حزمة تحتوي على الآلة الإفتراضية لجافا 7 لنظام أوبونتو :

```
openjdk-7-jre
```

وتختلف معماريتها حسب معمارية نظام أوبونتو، فإذا كان النظام هو 32 بت تكون حزمة جافا 32 بت، وإذا كان 64 بت تكون حزمة جافا 64 بت. لكن يمكن تثبيت جافا 32 بت في نظام أوبونتو 64 بت - كذلك في نظام وندوز- وذلك لأن بعض البرامج تتطلب جافا 32 بت، لكن لا يمكن تثبيت جافا 64 بت في نظام تشغيل 32 بت.

عند إنتاج برامج جافا يُمكن تشغيلها في أي نظام تشغيل مباشرة عند وجود الآلة الإفتراضية المناسبة، ولا يحتاج

البرنامج لإعادة ترجمة حتى يعمل في أنظمة غير النظام الذي تم تطوير البرنامج فيه. مثلاً يُمكن تطوير برنامج جافا في بيئة لينكس لإنتاج برامج يتم نقلها وتشغيلها مباشرة في وندوز أو ماكنتوش. وتختلف عنها لغة سي وأوبجكت باسكال في أنها تحتاج لإعادة ترجمة البرامج مرة أخرى في كل نظام تشغيل على حده قبل تشغيل تلك البرامج. لكن برامج لغة سي وأوبجكت باسكال لاتحتاج لآلة إفتراضية في أنظمة التشغيل بل تتعامل مع نظام التشغيل ومكتباته مباشرة.

# أدوات تطوير جافا Java SDK

آلة جافا الإفتراضية السابقة تُمكّن برامج جافا من العمل في نظام التشغيل، لكنها لا تحتوي على مترجم، لذلك لا يمكن كتابة برامج جافا وتطويرها بها، ولتطوير وترجمة وتنقيح برامج جافا وتحويلها إلى byte code لابد من الحصول على الـ SDK الخاص بالجافا، أي مايُعرف بالـ Java SDK. وهو يأتي في شكل برنامج للتثبيت به مترجم جافا (compiler)، ومنقح (debugger)، وآلة جافا الإفتراضية، أي لانحتاج لتثبيت آلة جافا الإفتراضية لوحدها عند تثبيت Java SDK. واسم الملف التالي يُمثل الـ Java SDK لبيئة وندوز:

```
jdk-7u51-windows-x64.exe
```

وهو مخصص لنظام وندوز 64 بت ويُمثّل جافا 7. ونُلاحظ أنه يبدأ بالإسم jdk وهو إختصار لـ Java Development Kit.

والملف التالي يُمثل حزمة Java SDK لنظام التشغيل أوبونتو:

```
openjdk-7-jdk
```

# بيئة التطوير NetBeans

وهي من أفضل بيئات التطوير للغة جافا، وقد تمت كتابتها باستخدام لغة جافا نفسها بواسطة شركة أوراكل صاحبة تلك اللغة.

يُمكن استخدام هذه الأداة لتطوير برامج بلغات برمجة أخرى غير الجافا مثل برامج PHP و سي++.

توجد أدوات تطوير أخرى مشهورة و هي Eclipse وهي مستخدمة من قبل مبرمجين كُثر، و أخرى تسمى IntelliJ والتي بُنيت عليها بيئة تطوير أندرويد.

جميع بيئات التطوير هذه تحتاج إلى تثبيت Java SDK أولاً قبل تثبيتها

## المؤلف: معتز عبدالعظيم

أعمل مطور برامج وكُنت فقط استخدم لغة أوبجكت باسكال كلغة برمجة أساسية، لكن منذ عام 2011 بدأت تعلم جافا وكتبت بها عدد من البرامج. وكان سبب تعلمي لها واعتمادي لها في تطوير كثير من البرامج هو:

1. أنه يوجد عدد كبير من المبرمجين يستخدمون لغة جافا، بل أن معظمهم درسها في الجامعة. لذلك يُمكن أن تكون لغة مشتركة بين عدد كبير من المبرمجين.
2. توجد مكتبات كثيرة ومجانية تدعم مجال الإتصالات مكتوبة بلغة جافا، وهو المجال الذي أعمل فيه.
3. أنها مجانية ويتوفر لها أدوات تطوير متكاملة ذات إمكانات عالية في عدد من المنصات. ماعلى المبرمج إلا إختيار المنصة المناسبة له
4. تدعم البرمجة الكائنية بصورة قوية
5. أن البرامج الناتجة عنها متعددة المنصات والمعماريات بمعنى الكلمة، ولايحتاج المُبرمج إنتاج عدد من الملفات التنفيذية لكل معمارية على حده. بل يحتاج لإنتاج ملف byte code واحد يكفي لمعظم المعماريات وأنظمة تشغيل الكمبيوتر المعروفة.
6. أداة التطوير Netbeans وطريقة تقسيم الحزم packages مناسبة للبرامج الكبيرة والتي تحتاج تقسيماً منطقياً والوصول لتلك الأقسام بسرعة وسهولة وتُسهل أيضاً التشارك في كتابة البرامج.

## ترخيص الكتاب



ملحوظة

**لا يُفضّل نسخ ثم اللصق في بيئة NetBeans من هذا الكتاب لأنه يتم أحياناً نقل أحرف غير مرئية تتسبب في تعثر ترجمة البرامج. لذلك من الأفضل كتابة الأمثلة يدوياً.**

# المحتويات

# البرنامج الأول.

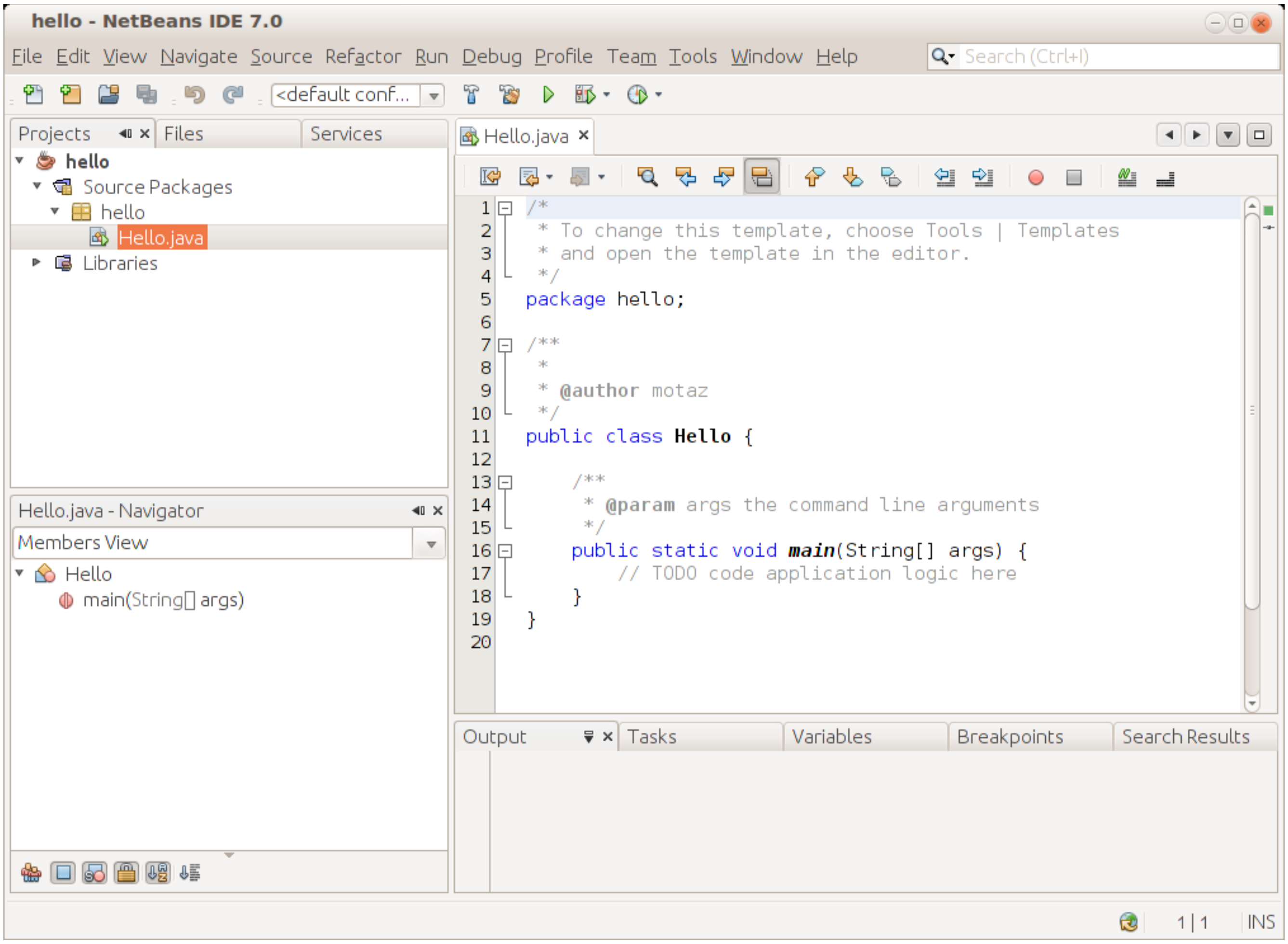
بعد تثبيت آلة جافا الإفتراضية وأداة التطوير NetBeans نقوم باختيار New/Project ثم Java/Java Application. ثم نقوم بتسمية البرنامج *hello* ليظهر لنا الكود التالي:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package hello;

/*
 *
 * @author motaz
 */
public class Hello {

    /*
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }
}
```

فإذا لم يظهر الكود نقوم بفتح الملف *hello.java* بواسطة شاشة المشروع التي تظهر يسار شاشة NetBeans كما في الشكل التالي:




بعد ذلك نقوم بكتابة السطر التالي داخل الإجراء main

```
        System.out.print("Hello Java world\n");
```

ليصبح الكود كالتالي:

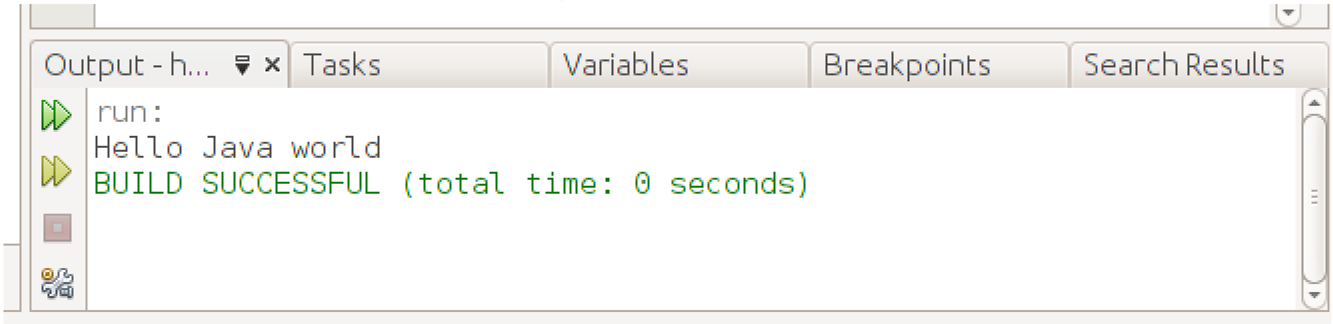
```
package hello;

/**
 *
 * @author motaz
 */
public class Hello {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.print("Hello Java world\n");

    }
}
```

يتم تشغيل البرنامج عن طريق المفتاح F6 ليظهر لنا المُخرجات في أسفل شاشة NetBeans




الأمر System.out.print يقوم بكتابة نص أو متغير في شاشة الطرفية.

الرمز \n مهمته هو الإنتقال للسطر الجديد في الطرفية، يمكن استخدام println والذي يقوم بالإنتقال للسطر الجديد دون الحاجة لإستخدام رمز السطر الجديد \n ليصبح الأمر كالتالي:

```
System.out.println("Hello Java world");
```

لتشغيل البرنامج الناتج خارج بيئة التطوير، نقوم أولاً ببناء الملف التفنيذي بواسطة Build وذلك بالضغط على المفاتيح Shift + F11 . بعدها نبحث عن الدليل الذي يحتوي على برامج NetBeans ويكون اسمه في الغالب *NetBeansProjects* ثم داخل الدليل *hello* نجد دليل اسمه dist يحتوي على الملف التنفيذي. في هذه الحالة يكون اسمه *hello.jar*

يُمكن تنفيذ هذا البرنامج في سطر الأوامر في نظام التشغيل بواسطة كتابة الأمر التالي:

```
java -jar hello.jar
```

يُمكن نقل هذا الملف التنفيذي من نوع Byte code إلى أي نظام تشغيل آخر يحتوي على آلة جافا الإفتراضية ثم تنفيذه بهذه الطريقة. ونُلاحظ أن حجم الملف التنفيذي صغير نسبياً (حوالي كيلو ونصف) وذلك لأننا لم نستخدم مكتبات إضافية.

بعد ذلك نقوم بتغيير الكود إلى التالي:

```
        int num = 9;
        System.out.print(num + " * 2 = " + num * 2 + "\n");
```

وهذه طريقة لتعريف متغير صحيح أسميناه *num* وأسندنا له قيمة إبتدائية 9

وفي السطر الذي يليه قُمنا بكتابة قيمة المتغير، ثم كتابة قيمته مضروبة في الرقم 2. وفي نهاية الإجراء أضفنا الرمز \n ومهمته هو الإنتقال إلى سطر الجديد في شاشة الطرفية. وهذا هو ناتج تشغيل البرنامج:

```
9 * 2 = 18
```

لطباعة التاريخ والساعة الحاليين نكتب هذه الأسطر:

```
        Date today = new Date();
        System.out.print("Today is: " + today.toString() + "\n");
```

ولابد من إضافة المكتبة المحتوية على الفئة Date في بداية البرنامج:

```
import java.util.Date;
```

فيصبح شكل كود البرنامج الكلي هو:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package hello;

import java.util.Date;


public class Hello {

    /*
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int num = 9;

        System.out.print(num + " * 2 = " + num * 2 + "\n");
        Date today = new Date();
        System.out.print("Today is: " + today.toString() + "\n");

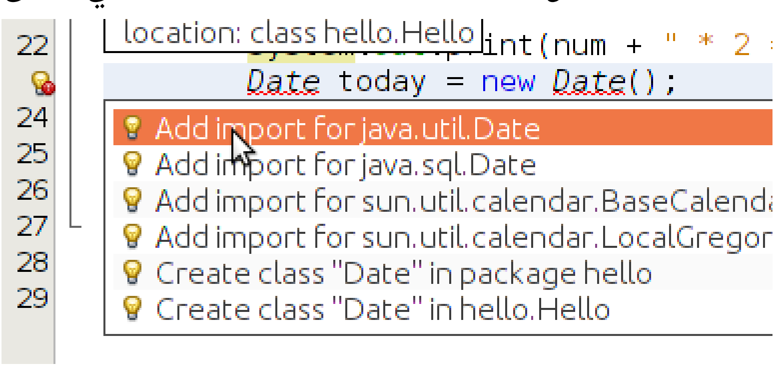

    }
}
```

وهذا هو ناتج تشغيل البرنامج

```
9 * 2 = 18
Today is: Fri Jul 31 11:59:46 EAT 2015
```

ملحوظة:

يُمكن إضافة إسم المكتبة تلقائياً عند ظهور العلامة الصفراء شمال السطر الموجودة فيه الفئة Class التي تحتاج لتلك المكتبة كما تظهر في هذه الصورة:



ثم اختيار *Add import for java.util.Date*

وهذه ميزة مهمة في أداة التطوير NetBeans تغني عن حفظ أسماء المكتبات المختلفة.

يمكن تغيير نسق التاريخ والساعة وذلك باستخدام الكائن SimpleDateTime كما في المثال التالي:

```
    SimpleDateFormat simpleFormat = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
    Date today = new Date();
    System.out.print("Today is: " + simpleFormat.format(today) + "\n")
```

والناتج هو:

```
Today is: 31.07.2015 12:03
```

ويمكن تغيير النسق بتغيير موضع الرموز التي ترمز لمكونات التاريخ وهي:

dd يُمثل اليوم

MM يُمثل رقم الشهر

yyyy يُمثل السنة كاملة، يمكن اختصارها في yy لتصبح رقمين فقط، مثلاً 15 والتي تعني 2015

HH: الساعة بنسق 24 ساعة

mm: الدقائق

ss: الثواني

وهذا مثال آخر لنسق مختلف

```
  SimpleDateFormat simpleFormat = new SimpleDateFormat("E dd.MMMM.yyyy hh:mm:ss a");
  Date today = new Date();
  System.out.print("Today is: " + simpleFormat.format(today) + "\n");
```

وهذا هو الناتج:

```
Today is: Fri 31.July.2015 12:10:21 PM
```

استخدمنا E لكتابة اليوم من الإسبوع، و MMMM لكتابة اسم الشهر كاملاً، ويمكن استخدام MMM لكتابة اسم الشهر بطريقة مختصرة، واستخدمنا h لكتابة الساعة بنسة 12 ساعة، ولا بد من استخدام a معها لتوضيح هل هو مساءً أم صباحاً am/pm

يمكن استخدام SimpleDateFormat لتحويل التاريخ من شكل نص String إلى تاريخ Date وذلك باستخدام الدالة parse لكن لابد من مطابقة النسق وإلا حدث خطأ:

```
   public static void main(String[] args) throws ParseException {

       String todayStr = "15.10.2012";
```

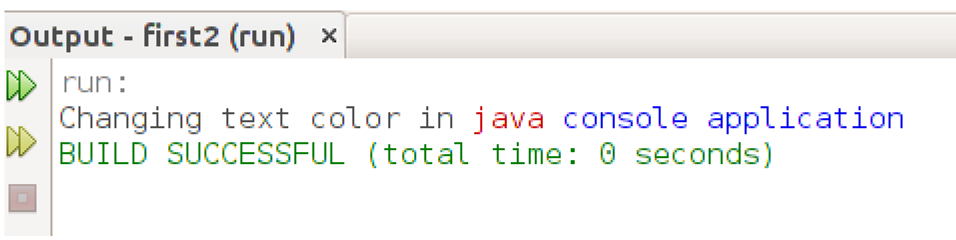
```java
        SimpleDateFormat simpleFormat = new SimpleDateFormat("dd.mm.yyyy");
        Date today = simpleFormat.parse(todayStr);
        System.out.print("Today is: " + simpleFormat.format(today) + "\n");
    }
```

نلاحظ أننا قمنا بإضافة *throws ParseException* في بداية الدالة *main* وذلك بعد أن أقترحت علينا بيئة التطوير هذه الإضافة وذلك لأن عملية التحويل هذه ربما ينتج عنها خطأ إذا كانت القيمة المدخلة غير صحيحة أو تحتوي على أحرف مثلاً أو قيم تاريخ غير صحيحة مثلاً تم إدخال 15 في خانة الشهر أو 32 خانة الأيام، أو ربما تم إدخال تاريخ بغير النسق، مثلاً 15/10/2012. وسوف نتكلم لاحقاً على معالجة الإستثناءات في لغة جافا في هذا الكتاب بإذن الله.

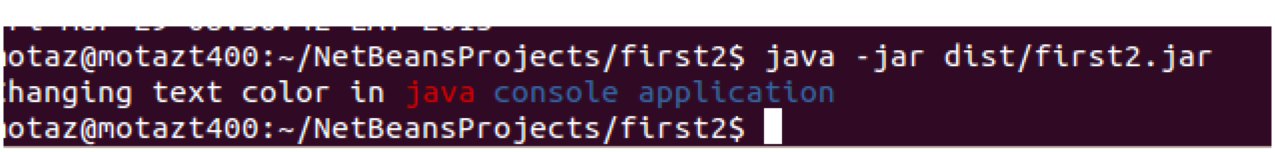
في المثال التالي قُمنا بتغيير لون جزء من النص بالطريقة التالية:

```java
        System.out.print("Changing text color in ");
        System.out.print("\033[31m"); // Change color to red
        System.out.print("java ");
        System.out.print("\033[34m"); // Change to blue
        System.out.print("console application");
        System.out.println("\033[0m"); // change to default color
```

فتظهر النتيجة بالشكل التالي في بيئة NetBeans:




وتظهر بالشكل التالي عند تنفيذ البرنامج من الطرفية:



# برنامج الواجهة رسومية

من الأشياء المهمة في أدوات التطوير و لغات البرمجة هو دعمها للواجهات الرسومية أو مايُسمى بال Widgets. كل نظام تشغيل يحتوي على مكتبة أو أكثر تُمثل واجهة رسومية، مثلاً يوجد في نظام لينكس واجهات GTK و QT و في نظام وندوز توجد مكتبة وندوز الرسومية، وفي نظام ماكنتوش توجد مكتبات Carbon و Cocoa. أما جافا فلها مكتباتها الخاصة والتي تعمل في كل هذه الأنظمة ومنها واجهة Swing.

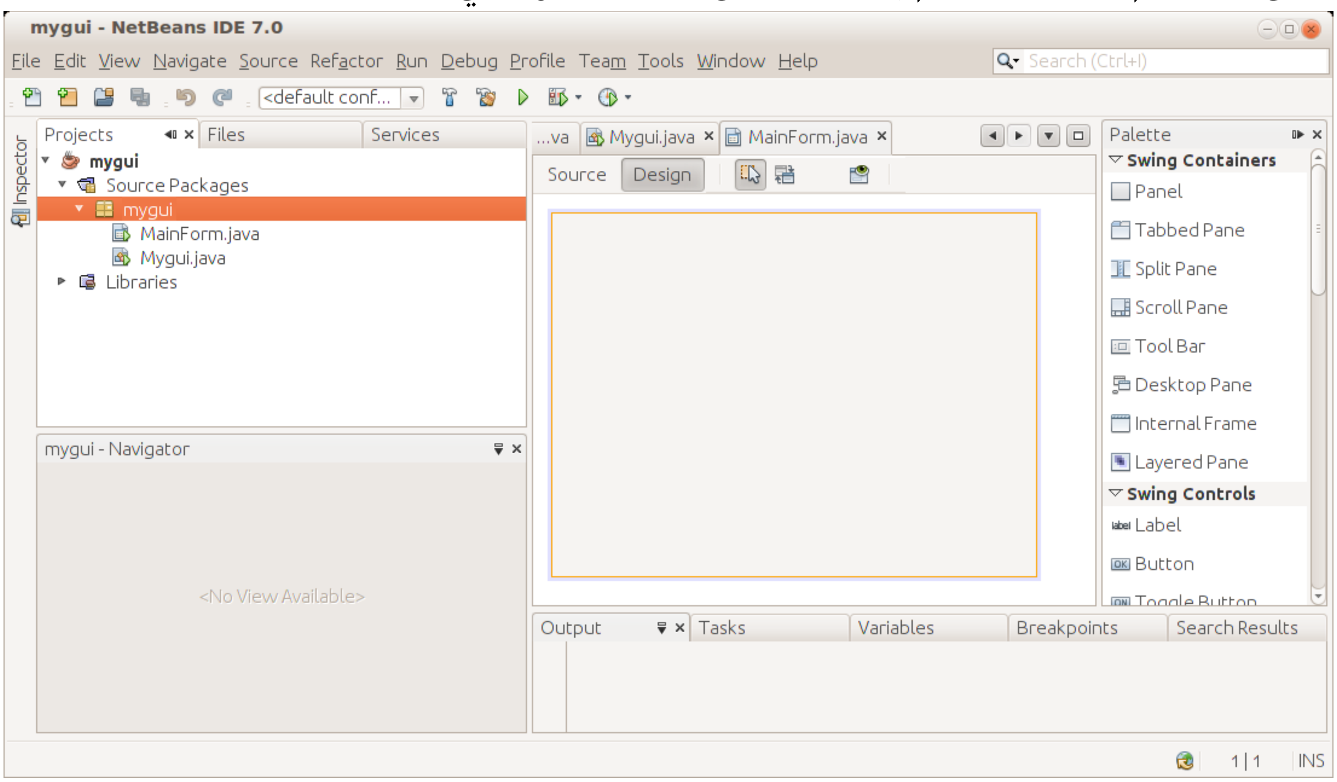
لكتابة أول برنامج ذو واجهة رسومية في جافا باستخدام NetBeans نختار File/New Project

ثم نختار  Java/Java Application

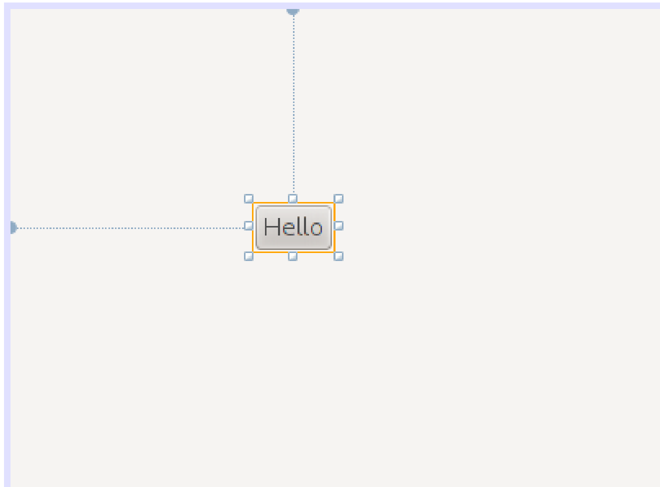
ونقوم بتسميته مثلاً mygui.

في شاشة Projects نختار الحزمة mygui ثم بالزر اليمين للماوس نختار New/JFrame Form

نسمى هذا الفورم MainForm فيتم إضافته للمشروع ويظهر بالشكل التالي:




يظهر الفورم الرئيسي المسمى MainForm.java في وسط الشاشة. وفي اليمين نلاحظ وجود عدد من المكونات في صفحة ال Palette. نقوم بإدراج زر Button في وسط الفورم الرئيسي، ثم نقوم بتغيير عنوانه إلى *Hello*، وذلك إما بالضغط على زر F2 ثم تغيير العنوان، أو بالنقر على الزر اليمين في الماوس في هذا الزر ثم نختار Properties ثم Text




نرجع مرة أخرى للخصائص لنضيف حدث عند الضغط على الزر. هذه المرة نختار Events ثم في الخيار actionPerformed نختار الحدث jButton1ActionPerformed بعدها يظهر هذا الكود في شاشة الـ Source:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
  }
```

أو يُمكن إظهار هذا الكود بواسطة النقر المزدوج على الزر double click

فنقوم بتكابة كود لإظهار عبارة (السلام عليكم) عند الضغط على هذا الزر. فيصبح الكود الحدث كالتالي:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        String msg = "السلام عليكم";
        JOptionPane.showMessageDialog(null, msg);
   }
```

نلاحظ أننا قُمنا بتعريفا المتغير *msg* من النوع المقطعي *String* ثم قُمنا بإسناد قيمة إبتدائية له: "*السلام عليكم*"

بعد ذلك نرجع للحزمة الرئيسية Mygui.java ثم نكتب الكود التالي في الإجراء main:

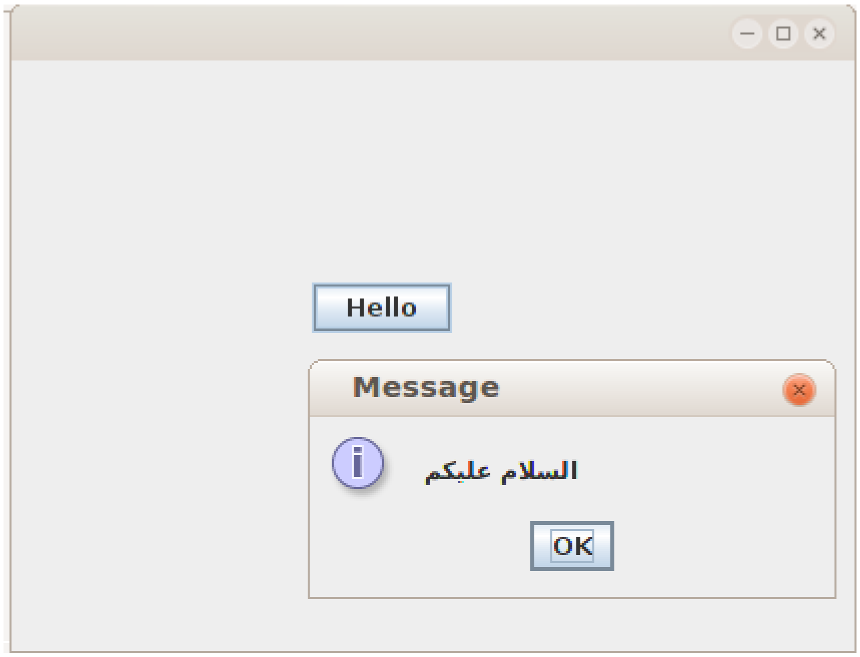
```
  public static void main(String[] args) {
      MainForm form = new MainForm();
      form.setVisible(true);
  }
```

في السطر الأول نُعرّف الكائن form من النوع MainForm الذي قُمنا بتصميمه، ثم نقوم بإنشاء نسخة من هذا النوع و تهيئته بواسطة

```
new MainForm
```

وفي السطر الثاني قمنا بإظهار الفورم في الشاشة.

عند تنفيذ البرنامج يظهر بالشكل التالي عند الضغط على الزر:




نرجع مرة أخرى للفورم في شاشة التصميم (Design) ونقوم بإدراج المكون TextField لتُدخل فيه إسم المستخدم، ثم مكون من نوع Label نكتب فيه كلمة (الإسم) ثم مكون آخر من نوع Label نقوم بتغير إسمه إلى *jlName* وذلك في فورم الخصائص في صفحة Code في قيمة Variable Name

ثم نُدرج زر نكتب فيه كلمة (ترحيب) كما في الشكل التالي:



في الحدث ActionPerfomed لهذا الزر الجديد (ترحيب) نكتب الكود التالي لكتابة إسم المستخدم في المكون jLabel2

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {

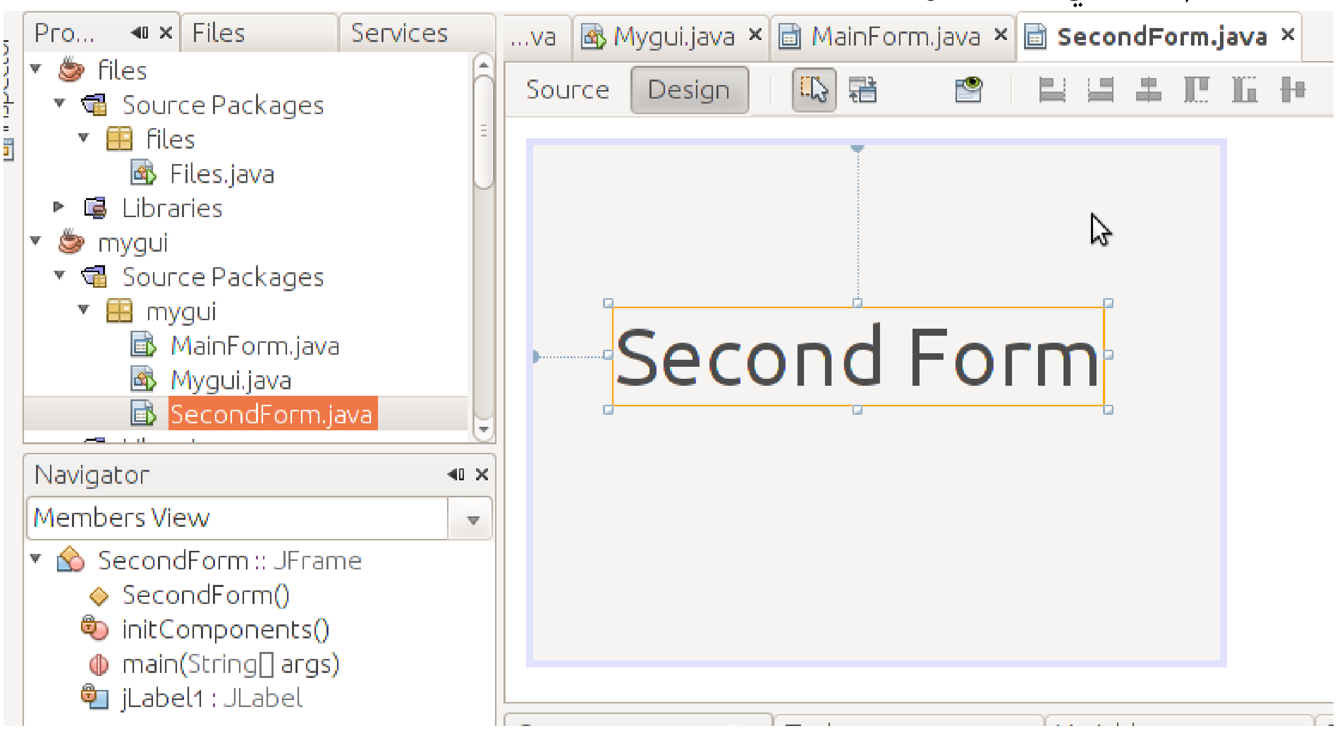
    jlName.setText("مرحباً بك " + jTextField1.getText());
}
```

نلاحظ أن الإجراء *getText* يُستخدم لقراءة محتويات الحقل النصي Text Field والإجراء *setText* يقوم بتغيير النص للمكون Label.

# الفورم الثاني

لإضافة وإظهار فورم ثاني في نفس البرنامج، نتبع الخطوات في المثال التالي:

نقوم بإضافة JFrame Form ونسميه *SecondFrom* ونضع فيه Label نكتب فيه عبارة "Second Form" ونزريد حجم الخط في هذا العنوان بواسطة Properties/Font.




في خصائص هذا الفورم الجديد نقوم بتغيير الخاصية *defaultCloseOperation* إلى *Dispose* بدلاً من *EXIT_ON_CLOSE* لأننا إذا تركناها في الخيار الأخير يتم إغلاق البرنامج عندما نغلق الفورم الثاني. وجرت العادة أن يتم إغلاق أي برنامج عند إغلاق شاشته الرئيسة. إغلاق الشاشات الفرعية يفترض به أن يقودنا إلى الشاشات الرئيسة.

نضيف زر في الفورم الرئيسي MainForm ونكتب الكود التالي في الحدث ActionPerformed في هذا الزر الجديد لإظهار الفورم الثاني، أو يمكن كتابة هذا الكود في زر الترحيب.

```
        SecondForm second = new SecondForm();
        second.setVisible(true);
```

يُمكن إرسال كائن أو متغير للفورم الجديد. مثلاً نريد كتابة رسالة الترحيب في الفورم الثاني.
لعمل ذلك نحتاج لتغير إجراء التهيئة constructor في الفورم الثاني والذي اسمه SecondForm ، نضيف إليه مدخلات:

```
    public SecondForm(String atext) {
        initComponents();
        jLabel1.setText(atext);
    }
```

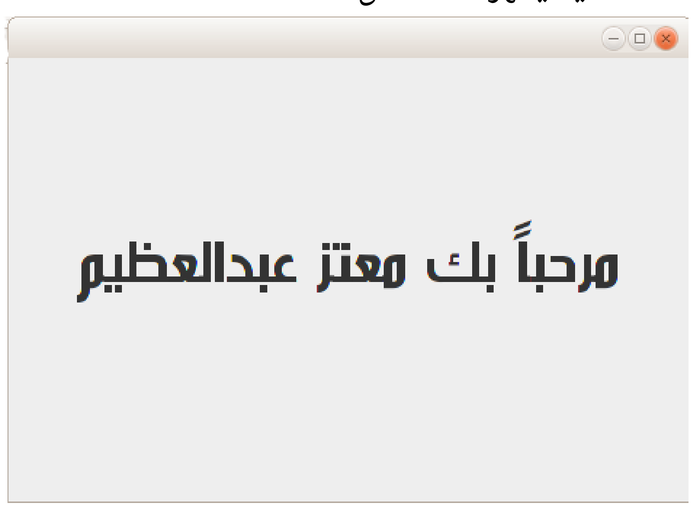
ثم نظهر هذه المدخلات – والتي هي عبارة عن رسالة الترحيب – في العنوان jLabel1
وعند تهيئة الفورم الثاني من الفورم الرئيسي نقوم بتعديل إجراء التهيئة إلى الكود التالي، وهذا الكود كتبناه في إجراء زر الترحيب:

```
   private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
       jlName.setText("مرحباً بك " + jTextField1.getText());

       SecondForm second = new SecondForm(jlName.getText());
       second.setVisible(true);
   }
```

عند التنفيذ يظهر هذا الشكل:



# الملفات

التعامل مع الملفات من الأشياء الأساسية في أي لغة برمجة وفي أي نظام تشغيل. و كثير من البرامج تحتاج لتخزين بعض البيانات في ملفات على القرص، أو قراءتها، وتشمل العمليات على الملفات: إنشاء ملف جديد، قراءة محتويات ملف، حذف ملف أو التأكد من وجود ملف في مسار معين.

في المثال التالي سوف نقوم بإختبار وجود الملف *myfile.txt* وإذا لم يكن موجود سوف يقوم البرنامج بإنشاء ملف جديد بهذا الإسم:

```
public static void main(String[] args) throws IOException {

    File file = new File("/home/motaz/myfile.txt");
    if (file.exists()) {
        System.out.println("File exists");
    }
    else {
        System.out.println("File does not exist");
        file.createNewFile();
    }
}
```

نلاحظ أننا استخدمنا عبارة *throws IOException* في نهاية الدالة الرئيسية *main* وذلك لاحتمال حدوث خطأ أثناء إنشاء الملف، مثلاً قد يكون المسار المحدد هو للقراءة فقط، أو ليس للمستخدم الحالي صلاحية لكتابة ملف في هذا المسار.

كذلك استخدمنا النوع *File* وقمنا بتعريف كائن منها هو *file* وذلك لغرض ربط البرنامج بالملف الخارجي على القرص. ويمكن عمل عدة عمليات للملف مثل الحذف *file.delete* او الإنشاء *file.createNewFile* أو التأكد من وجود الملف *file.exists*

هذا المثال تمت كتابته في بيئة لينكس، يمكن تغيير المسار بما يناسب نظام التشغيل، مثلاً في وندوز يمكن أن يكون c:\directory\myfile.txt

بدلاً من استخدام عبارة throws IOException كان من الممكن عمل معالجة للأخطاء وذلك بالطريقة التالية:

```
public static void main(String[] args) {

    try {
        File file = new File("/home/motaz/myfile.txt");
        if (file.exists()) {
            System.out.println("File exists");
        }
        else {
            System.out.println("File does not exist");
            file.createNewFile();
```

```
            }
        }
        catch (Exception ex){
                System.err.println("Unable to create file: " +
                        ex.toString());
        }
    }
```

فإذا حدث أي خطأ بعد عبارة *try* يتم تحويل التنفيذ إلى جزء *catch*. وهي طريقة أفضل لإظهار المشكلة كما يريدها المبرمج للمستخدم، بدلاً من ترك المترجم يكتب رسالة الخطأ مباشرة للمستخدم.

هذه هي طريقة حماية أي جزء من الكود والذي يمكن أن يكون عرضة للأخطاء أثناء التشغيل:

```
    try{
      // الكود المعرض لأخطاء التشغيل

       return (true);

     }
     catch (Exception e)
     {
       System.err.println("Error: " + e.getMessage());
       return (false); // fail
     }
```

# كتابة نص في ملف

توجد عدة طرق للكتابة أو لقراءة ملف نصي، اخترنا في هذه الأملثة أحد هذه الطرق.

في المثال التالي نُريد الكتابة في ملف نصي باستخدام برنامج بدون واجهة رسومية (console application) وذلك باختيار Java/Java Application.

هذه المرة نُريد كتابة إجراء جديد نعطيه إسم الملف المُراد إنشاءه والكتابة فيه والنص الذي نُريد كتابته في هذا الملف.

قمنا بتسمية المشروع *files*، وكتبنا الإجراء الجديد أسفل الإجراء *main* الموجود مسبقاً. وأسمينا الإجراء الجديد *writeToTextFile* وعرفناه بهذه الطريقة:

```
private static boolean writeToTextFile(String aFileName, String text)
{

}
```

نلاحظ أننا قُمنا بتعريف مُدخلين لهذا الإجراء وهما *aFileName* وهو من النوع النصي ليستقبل إسم الملف المراد كتابته، والآخر *text* وهو من النوع النصي أيضاً والذي يُمثل المحتويات المُراد كتابتها في الملف.

ثم نقوم بكتابة الكود التالي داخل هذا الإجراء:

```
private static boolean writeToTextFile(String aFileName, String text)
{
    try{

      File file = new File(aFileName);
      FileWriter writer = new FileWriter(file);

      writer.write(text + "\n");
      writer.close();
      return (true); // success

     }
       catch (Exception e)
     {
       System.err.println("Error: " + e.getMessage());
       return (false); // fail
     }

}
```

نلاحظ أننا قُمنا بإرجاع القيمة *true* في حال أن الكتابة في الملف تمت بدون حدوث خطأ. أما في حالة حدوث

الخطأ قمنا بإرجع القيمة *false* وذلك ليعرف من يُنادي هذا الإجراء أن العملية نجحت أم لا.

بالنسبة لتعريف الملف وتعريف طريقة الكتابة عليه قُمنا بكتابة هذه الأسطر الثلاثة:

```
    File file = new File(aFileName);
    FileWriter writer = new FileWriter(file);
```

في العبارة الأولى قُمنا بتعريف الكائن *file* من نوع الفئة *File* وهو كائن للربط مع الملف الخارجي. وقد أعطيناه إسم الملف في المدخلات. وفي العبارة الثانية قُمنا بتعريف الكائن *writer* من النوع *FileWriter* وذلك للكتابة على الملف، ومُدخلاته هو الكائن *file*.

بعد ذلك قُمنا بكتابة النص المُرسل داخل الملف باستخدام الكائن *writer* بالطريقة التالية:

```
    writer.write(text + "\n");
```

في النهاية قمنا بإغلاق الملف باستخدام عبارة *writer.close* وهي من الأهمية بمكان بحيث أنه يمنع برنامج آخر بالكتابة على هذا الملف الذي لم يتم إغلاقه، وكذلك فإن الملف غير المغلق يمكن أن يتسبب في إهدار للموارد، حيث أن نظام التشغيل يسمح بفتح عدد معين من الملفات في آن واحد، فتكرار عملية فتح الملف دون أن يكون هناك إغلاق له يمكن أن يمنع فتح ملفات جديدة أثناء تشغيل البرنامج.

ولنداء هذا الإجراء يجب إستدعاءه من الإجراء الرئيسي main بالطريقة التالي:

```
     writeToTextFile("myfile.txt", "my text");
```

ويُمكن تحديد المسار أو الدليل الذي نُريد كتابة الملف عليه كما فعلنا في المثال التالي لنداء هذا الإجراء. وقد قُمنا بإضافة التاريخ والوقت الذي تمت فيه كتابة الملف:

```
    public static void main(String[] args) {
        // TODO code application logic here
        Date now = new Date();
        boolean result;
        result = writeToTextFile("/home/motaz/java.txt",
                "This file has been written using Java\n" + now.toString());
        if (result){
            System.out.print("File has been written successfully\n");
        }
        else{
            System.out.print("Error has occurred while writing in the file\n");
        }

    }
```

كذلك فقد قُمنا بتعريف المتغير *result* من النوع المنطقي *boolean* والذي يحتمل فقط القيم *true/false* وذلك لإرجاع نتيجة العملية هل نجحت أم لا.

وقد قُمنا بفحص قيمة المتغير *result* لعرض رسالة تُفيد بأن العلمية نجحت، أو فشلت في حالة أن قيمته *false*.
و العبارة الشرطية هي *if*

```
  if (result)
```

معناها أن قيمة *result* إذا كانت تحمل القيمة *true* قم بتنفيذ العبارة التالية، أما إذا لم تكن تحمل تلك القيمة فقم بتنفيذ الإجراء بعد الكلمة *else*

لتنفيذ هذا البرنامج نحتاج لإضافة المكتبات التالية، والتي تساعد أداة التطوير في إضافتها تلقائياً:

```
import java.io.File;
import java.io.FileWriter;
import java.util.Date;
```

بدلاً من حذف محتويات الملف في كل مرة، يمكن الإضافة فقط في النهاية بما يعرف بمصطلح *append* وهو يعني الإضافة في نهاية الملف. لعمل ذلك نقوم بتغيير طريقة تهيئة الكائن *writer* وذلك بإضافة المُدخل *true* كالتالي:

```
      FileWriter writer = new FileWriter(file, true);
```

فعند تشغيله أكثر من مرة، نلاحظ أن المحتويات القديمة موجودة وأن الإضافة تتم في النهاية.

# القراءة من ملف

للقراءة من ملف يُمكن استخدام النوع FileReader لقراءة محتويات أي نوع من الملفات. وقد قمنا بعمل التغييرات اللازمة لتحويل محتويات الملف إلى نص كما في المثال التالي:

```
private static boolean readTextFile(String aFileName)
{
   try {

     File file = new File(aFileName);
     FileReader reader = new FileReader(file);

     char buf[] = new char[10];
     int numread;
     while ((numread=reader.read(buf)) > 0) {

         String text = new String(buf, 0, numread);
         System.out.print(text);
     }
     reader.close();
     return (true); // success

    }
    catch (Exception e)
    {
      System.err.println("Error: " + e.getMessage());
      return (false); // fail
    }

}
```

نلاحظ أننا استخدمنا سلسلة من النوع char وهو يقوم بتخزين رمز، والنصوص هي مجموعة من الرموز.

```
char buf[] = new char[10];
```

لقراءة كل محتويات الملف، لابد من قراءة جميع الأحرف، في كل مرة نقوم بقراءة 10 أحرف على الأكثر إلى أن تنتهي محتويات الملف. استخدمنا العبارة التالية لقراءة جزء من الملف ثم نقوم باختبار هل وصل الملف إلى نهايته أم لا:

```
while ((numread=reader.read(buf)) > 0) {
```

في هذ الجزء يقوم البرنامج بقراءة محتويات الملف وقوم بتخيزنها في السلسلة *buf* وبما أن حجمها هو 10 بايت فيقوم بمحاولة قراءة 10 رموز أو أحرف من الملف، ثم يقوم بإرجاع العدد الذي قرأه في المتغير *numread*، وفي نهاية الملف يمكن أن يتبقى جزء أقل من 10 أحرف، فبدلاً من إرجاع 10 يقوم بإرجاع ماتبقى مثلاً 5 أحرف. كذلك

فإن البرنامج في نفس السطر يقوم بمقارنة قيمة *numread* هل هي أكبر من الرقم 0 والتي تعني أنه نجح في قراءة بايت على الأقل، أما إذا كانت النتيجة صفراً فهي تعني أنه لم يتبقى مقطع للقراء في الملف فيخرج تنفيذ البرنامج من حلقة while.

بعد ذلك قمنا بتحويل سلسلة الأحرف إلى مقطع لسهولة التعامل معه وكتابته في الشاشة:

```
String text = new String(buf, 0, numread);
```

في كل الأحوال فإن طول السلسة buf هو 10 بايت، لكن ربما قرأ البرنامج عدداً أقل من الأحرف، لذلك نقوم بنسخ الجزء الذي تمت قراءته فعلياً لذلك قمنا بتحديد المقطع المراد قرائته بواسطة المُدخلات *0*, *numread*

نفرض أن الملف يحتوي على 25 رمزاً، فتكون القراءة كالتالي: في الدورة الأولى تتم قراءة 10 رموز، ثم في الدورة الثانية 10 رموز ثم 5 رموز. هذه الرموز تُمثل أحرف و رمز السطر الجديد المعروف بالـ new line/line feed في وندوز يتم استخدام رمزين للدلالة على نهاية السطر، أما في نظام لينكس فيتم استخدام رمز واحد فقط وهو new line. قمنا بكتابة رمز السطر الجديد في المثال السابق (الكتابة في ملف نصي) وذلك باستخدام

```
\n
```

لهذا السبب استخدمنا *print* بدلاً من *println* وذلك لأن النص المقروء من الملف يحتوي على رمز السطر الجديد بعد نهاية كل سطر، أما إذا استخدمنا *println* فسوف يتم الإنتقال إلى سطر جديد بعد كتابة كل 10 أحرف فتصبح الجُمل مقطعة كالتالي:

```
This file
has been w
ritten usi
ng Java
Fr
i Aug 28 0
9:20:47 EA
T 2015
```

لكن عند استخدام *print* يظهر النص واضحاً كالتالي:

```
This file has been written using Java
Fri Aug 28 09:20:47 EAT 2015
```

يُمكن تحويل كود القراءة في هذا الإجراء لأن يقوم البرنامج بقراءة محتويات الملف سطراً سطراً بدلاً من قراءة عدد من الرموز ثم تحويلها إلى مقطع *String*:

هذه المرة استخدمنا النوع *FileInputStream* والنوع *DataInputStream* والنوع *InputStreamReader* والنوع *BufferedReader* وذلك لقراءة سطراً كاملاً في كل مرة كالتالي:

```
    private static boolean readTextFile(String aFileName)
    {
```

```java
    try{
      FileInputStream fstream = new FileInputStream(aFileName);

      DataInputStream textReader = new DataInputStream(fstream);
      InputStreamReader reader = new InputStreamReader(textReader);
      BufferedReader lineReader = new BufferedReader(reader);

      System.out.print("Reading " + aFileName + "\n-------------\n");

      String line;

      while ((line = lineReader.readLine()) != null){
          System.out.println (line);
      }
      fstream.close();
      return (true); // success

    }
    catch (Exception ex)
    {
      System.err.println("Error in readTextFile: " + ex.getMessage());
      return (false); // fail
    }

}
```

قُمنا بنداء الإجراء الجديد من داخل main. ليصبح الإجراء كاملاً هو:

```java
  public static void main(String[] args) {
      // TODO code application logic here
      Date now = new Date();
      boolean result;
      result = writeToTextFile("/home/motaz/java.txt",
              "This file has been written\n using Java\n" + now.toString());
      if (result){
          System.out.print("File has been written successfully\n");
      }
      else {
          System.out.print("Error has occured while writing in the file\n");
      }

      readTextFile("/home/motaz/java.txt");
  }
```

# تعريف الكائنات والذاكرة

من الأمثلة السابقة نُلاحظ أننا استخدمنا البرمجة الكائنية في قراءة وكتابة الملفات والتاريخ. ونُلاحظ أن تعريف الكائن وتهيته يمكن أن تكون في عبارة واحدة، مثلاً لتعريف التاريخ ثم تهيئته بالوقت الحالي استخدمنا:

```
        Date today = new Date();
```

وكان يُمكن فصل التعريف للكائن الجديد من تهيئته بالطريقة التالية:

```
        Date today;
        today = new Date();
```

هذه المرة في العبارة الأولى قُمنا بتعريف الكائن *today* من نوع الفئة *Date*. لكن إلى الآن لا يُمكننا استخدام الكائن *today* فلم يتم حجز موقع له في الذاكرة.

أما في العبارة الثانية فقد قُمنا بحجز موقع له في الذاكرة بإستخدام الكلمة *new* ثم تهيئة اللكائن بإستخدام الإجراء

```
    Date();
```

والذي بدوره يقوم بقراءة التاريخ والوقت الحالي لإسناده للكائن الجديد *today*. وهذا الإجراء يُسمى في البرمجة الكائنية constructor.

في هذا المثال Date هي عبارة عن فئة لكائن أو تُسمى class في البرمجة الكائنية. و المتغير today يُسمى كائن object أو instance ويُمكن تعريف أكثر من كائن instance من نفس الفئة لاستخدامها. وتعريف كائن جديد من فئة ما وتهيئتها تُسمى object instantiation في البرمجة الكائنية.

بعد الفراغ من استخدام الكائن يمكننا تحريره من الذاكرة وذلك باستخدام الدالة التالية:

```
   today = null;
```

توجد في لغة جافا ما يُعرف بال garbage collector وهي آلية لحذف الكائنات الغير مستخدمة من الذاكرة تلقائياً عندما ينتهي تنفيذ الإجراء. يتم فقط حذف الكائنات المعرفة في نطاق هذا الإجراء. في معظم الأحيان لا نحتاج لاستخدام هذه العبارة، فإذا تم الإنتهاء من استخدام المتغير الذي يؤشر لهذا الكائن يتم تحريره تلقائياً.

مفهوم – غير مستخدم- يعني أنه لا يوجد مؤشر له من المتغيرات، حيث يمكن أن يكون لكائن ما عدد من المؤشرات تؤشر له، فعندما تنتهي جميع هذه المؤشرات ويصبح عدد المتغيرات التي تؤشر لهذا الكائن في الذاكرة صفراً يقوم الـ garbage collector بحذفه من الذاكرة بعد مدة معينة. أما لغات البرمجة الأخرى مثل سي وأوبجكت باسكال فعند استخدامها لابد من تحرير الكائنات يدوياً في معظم الحالات.

نهاية المتغير يكون بنهاية تنفيذ الحيز الموجود فيه وهو المحاط بالقوسين {}

نأخذ هذا المثال لشرح مفهوم حيز أو نطاق تعريف المتغير:

```
1     public static void main(String[] args) {
2
3        String yourName = "Mohammed";
4         {
5           String myName = "Motaz";
6           System.out.println(myName);
7         }
8         System.out.println(yourName);
9
10     }
```

نجد أن المتغير *yourName* معرف داخل الإجراء main لذلك لا ينتهي إلا بانتهاء هذا الإجراء، أي عند السطر رقم 10.

أما المتغير *myName* والمعرف في نطاق أضيق، فينتهي عند السطر رقم 7، فإذا أردنا أن نجعله ذو عمر أطول يمكن تعريفه خارج هذا النطاق الضيق:

```
    public static void main(String[] args) {

       String yourName = "Mohammed";
       String myName;
       {
          myName = "Motaz";
          System.out.println(myName);
        }
        System.out.println(yourName);

         readTextFile("/home/motaz/java.txt");
     }
```

بهذه الطريقة يصبح عمر المتغير myName مرتبط بنهاية الإجراء main، ونلاحظ أننا قمنا فقط بنقل التعريف إلى الخارج، لكن التهيئة فمازالت داخل ذلك الحيز، لكن هذا لم يؤثر على قيمته أو نطاق تعريفه.

يمكن تهيئة كائن جديد بواسطة إسناد مؤشر كائن قديم له، في هذه الحالة يكون كلا المتغيرين يؤشران لنفس الكائن في الذاكرة:

```
       Date today;
       Date today2;
```

```
        today = new Date();
        today2 = today;
        today = null;
        System.out.print("Today is: " + today2.toString() + "\n");
```

نلاحظ أننا لم نقم بتهيئة المتغير *today2* لكن بدلاً من ذلك جعلناه يؤشر لنفس الكائن *today* الذي تمت تهيئته من قبل.

بعد ذلك قُمنا بتحرير المتغير *today*، إلا أن ذلك لم يؤثر على الكائن، حيث أن الكائن مايزال مرتبط بالمتغير *today2*. ولا تقوم آلية garage collector بتحرير الكائن من الذاكرة إلا عندما تصبح عدد المتغيرات التي تؤشر له صفراً. فإذا قُمنا بتحرير المتغير today2 أيضاً تحدث مشكلة عند تنفيذ السطر الأخير، وذلك لأن الكائن تم تحريره من الذاكرة ومحاولة الوصول إليه بالقراءة أو الكتابة ينتج عنها خطأ.

ولمعرفة ماهو الخطأ الذي ينتج قُمنا بإحاطة الكود بعبارة try catch كما في المثال التالي:

```
        try {
          Date today;
          Date today2;
          today = new Date();
          today2 = today;
          today = null;
          today2 = null;
          System.out.print("Today is: " + today2.toString() + "\n");
        } catch (Exception e) {
            System.out.print("Error: " + e.toString() + "\n");
        }
```

والخطأ الذي تحصلنا عليه هو:

```
java.lang.NullPointerException
```

*ملاحظة:*

في لغة جافا أُصطلح على تسمية الفئات classes بطريقة أن يكون الحرف الأول كبير capital مثل Date, String, حتى الفئات التي يقوم المبرمج بكتابتها. أما الكائنات objects/instances فتبدأ بحرف صغير وذلك للتفرقة بين الفئة والكائن، مثل today, today2, myName.

# برنامج اختيار الملف

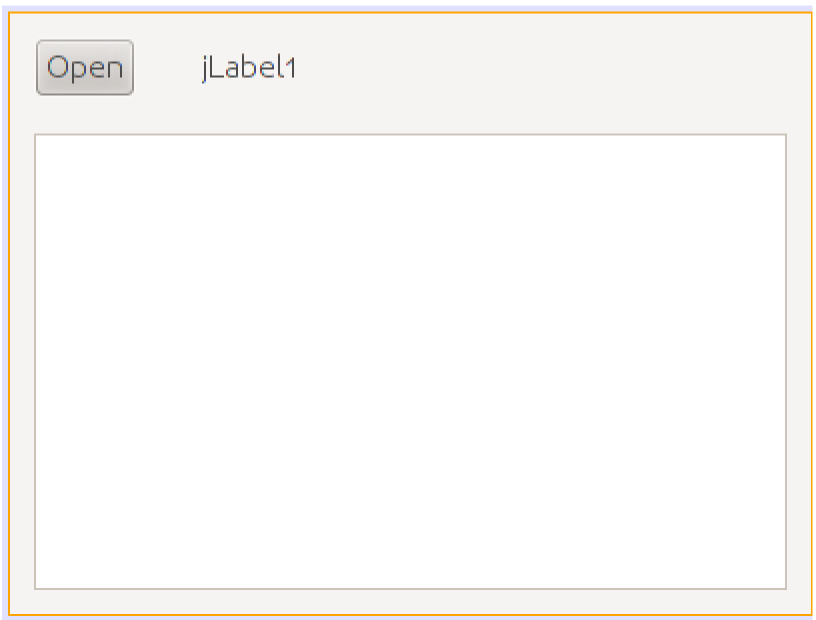
هذه المرة نُريد عمل برنامج ذو واجهة رسومية يسمح لنا بإختيار الملف بالماوس، ثم عرض محتوياته في صندوق نصي.

لعمل هذا البرنامج نفتح مشروع جديد بواسطة Java/Java Application. نسمي هذا المشروع *openfile*

نُضيف JFrame Form نسميه MainForm ونضع فيه المكونات التالية:

Button, Label, Text Area

كما في الشكل التالي:




بعد ذلك نكتب هذا الكود في الإجراء main في ملف البرنامج الرئيسي Openfile.java لإظهار الفورم فور تشغيل البرنامج:

```
    public static void main(String[] args) {

        MainForm form = new MainForm();
        form.setVisible(true);
    }
```

نقوم بنسخ الإجراء readTextFile من البرنامج السابق إلى كود البرنامج الحالي، ونعدله قليلاً، نضيف له مدخل جديد من نوع JTextArea وذلك لكتابة محتويات الملف في هذا المربع النصي بدلاً من شاشة سطر الأوامر Console. وهذا هو الإجراء المعدل:

```
private static boolean readTextFile(String aFileName, JTextArea textArea)
{
 try {
   FileInputStream fstream = new FileInputStream(aFileName);

   DataInputStream textReader = new DataInputStream(fstream);

   InputStreamReader isr = new InputStreamReader(textReader);
   BufferedReader lineReader = new BufferedReader(isr);

   String line;
   textArea.setText("");

   while ((line = lineReader.readLine()) != null){
     textArea.append(line + "\n");
   }

   fstream.close();
   return (true); // success


  }
  catch (Exception e)
  {
    textArea.append("Error in readTextFile: " + e.getMessage() + "\n");
    return (false); // fail
  }
```

وفي الحدث ActionPerformed في الزر نكتب الكود التالي:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

 final JFileChooser fc = new JFileChooser();
 int result = fc.showOpenDialog(null);
 if (result == JFileChooser.APPROVE_OPTION) {
        jLabel1.setText(fc.getSelectedFile().toString());
        readTextFile(fc.getSelectedFile().toString(), jTextArea1);
  }
}
```

وقد قُمنا بتعريف كائن اختيار الملف في السطر التالي:

```
 final JFileChooser fc = new JFileChooser();
```

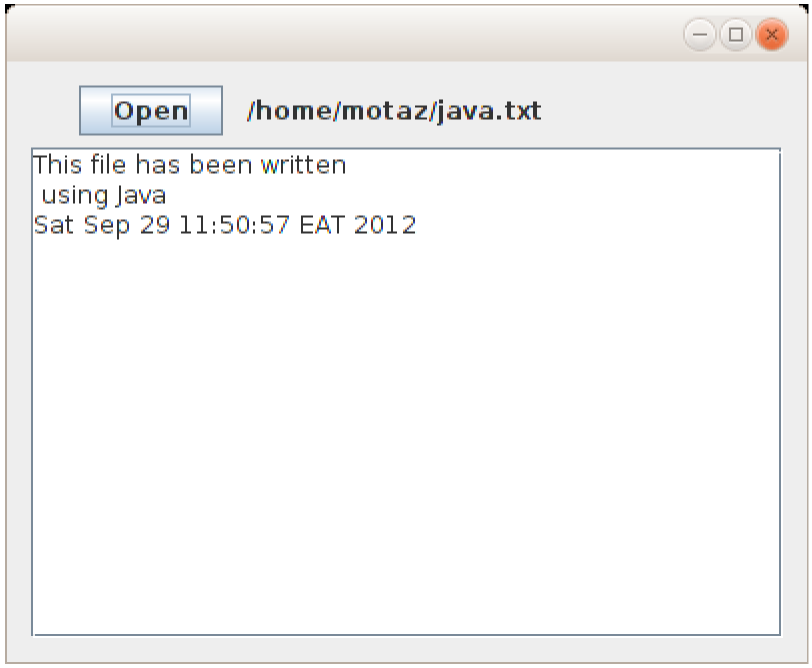
ثم قمنا بإظهاره ليختار المستخدم الملف في السطر التالي. ويقوم بإرجاع النتيجة: هل قام المستخدم باختيار ملف أم ضغط إلغاء:

```
 int result = fc.showOpenDialog(null);
```

فإذا قام باختيار ملف نقوم بكتابة اسمه في العنوان *jLabel1* ثم نظهر محتوياته داخل مربع النص:

```
    if (result == JFileChooser.APPROVE_OPTION) {
            jLabel1.setText(fc.getSelectedFile().toString());
            readTextFile(fc.getSelectedFile().toString(), jTextArea1);
     }
```

وعند تشغيل البرنامج يظهر لنا بهذا الشكل بعد إختيار الملف:




## كتابة فئة كائن جديد New Class

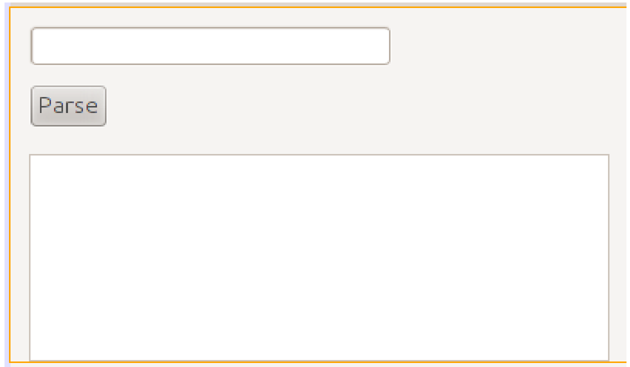
لغة جافا تعتمد فقط نموذج البرمجة الكائنية Object Oriented paradigm، وقد مر علينا في الأمثلة السابقة استخدام عدد من الكائنات، سواءً كانت لقراءة التاريخ أو للتعامل مع الملفات أو الكائنات الرسومية مثل Label وال Text Area والفورم JFrameForm. لكن حتى تصبح البرمجة الكائنية أوضح لابد من إنشاء فئات classes جديدة بواسطة المبرمج لتعريف كائنات منها.

في هذا المثال سوف نقوم بإضافة فئة class جديدة نُدخل لها جملة نصية لإرجاع الكلمة الأولى والأخيرة من الجملة.

قمنا بفتح برنامج جديد من نوع Java Application، و أسميناه newclass،

بعد ذلك أضفنا MainForm من نوع JFrameForm

ثم قُمنا بإدراج Text Field و Button و Text Area بهذا الشكل في الفورم الرئيسي:




ولا ننسى تعريف الفورم وتهيئته لإظهاره مع تشغيل البرنامج في الإجراء الرئيسي في الملف Newclass.java:

```
    public static void main(String[] args) {

        MainForm form = new MainForm();
        form.setVisible(true);
    }
```

بعد ذلك قمنا بإضافة class جديدة وذلك بإختيار Source Packages/new class بالزر اليمين ثم اختيار New/Java Class من القائمة. ثم نسمي الفئة الجديدة Sentence فيظهر لنا هذا الكود:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package newclass;

/*
 *
 * @author motaz
 */
public class Sentence {

}
```

وفي داخل كود الفئة -بين القوسين المعكوفين {} - قمنا بإضافة متغير مقطعي اسميناه *mySentence* لنحفظ فيه الجملة التي يتم إرسالها لتكون محتفظة بقيمة الجملة طوال فترة حياة الكائن.

ثم أضفنا الإجراء الذي يُستخدم في تهيئة الكائن، ولابد أن يكون اسمه مطابق لإسم الفئة:

```
    String mySentence;

    public Sentence (String atext){
       super();
       mySentence = atext;
    }
```

نلاحظ أنه في هذا الإجراء تم إسناد قيمة المُدخل *atext* إلى المتغير *mySentence* المُعرف على نطاق الكائن. حيث أن المتغير *atext* نطاقه فقط الإجراء *Sentence* وعند الإنتهاء من نداء هذا الإجراء يصبح غير معروف. لذلك إحتفظنا بالجملة المُدخلة في متغير في نطاق أعلى لتكون حياته أطول، حيث يُمكن استخدامه مادام الكائن لم يتم حذفه من الذاكرة.

بعد ذلك قُمنا بإضافة إجراء جديد في نفس فئة الكائن اسمه *getFirst* وهو يقوم بإرجاع الكلمة الأولى من الجملة:

```
public String getFirst(){
    String first;
    int firstSpaceIndex;
    firstSpaceIndex = mySentence.indexOf(" ");

    if (firstSpaceIndex == -1)
        first = mySentence;
    else
        first = mySentence.substring(0, firstSpaceIndex);

    return (first);
}
```

نلاحظ اننا استخدمنا الإجراء *indexOf* في المتغير أو الكائن المقطعي *mySentence* وقُمنا بإرسال مقطع يحتوي على مسافة. وهذا الإجراء أو الدالة مفترض به في هذه الحالة أن يقوم بإرجاع موقع أول مسافة في الجملة، وبهذه الطريقة نعرف الكلمة الأولى، حيث أنها تقع بين الحرف الأول وأول مسافة.

أما إذا لم تكن هُناك مسافة موجودة في الجملة فتكون نتيجة الدالة *indexOf* يساوي 1- وهذا يعني أن الجملة تتكون من كلمة واحدة فقط، في هذه الحالة نقوم بإرجاع الجملة كاملة (الجملة = كلمة واحدة).

وإذا وُجدت المسافة فعندها نقوم بنسخ مقطع من الجملة بإستخدام الدالة substring والتي نُعطيها بداية ونهاية المقطع المُراد نسخه. ونتيجة النسخ ترجع في المتغير أو الكائن المقطعي *first*

الدالة أو الإجراء الآخر الذي قُمنا بإضافته في الفئة *Sentence* هو *getLast* وهو يقوم بإرجاع آخر كلمة في الجملة:

```
public String getLast(){
    String last;
    int lastSpaceIndex;
    lastSpaceIndex = mySentence.lastIndexOf(" ");

    if (lastSpaceIndex == -1){
        last = mySentence;
    }
    else {
        last = mySentence.substring(lastSpaceIndex + 1, mySentence.length());
    }
    return (last);
}
```

وهو مشابه للدالة الأخرى، ويختلف في أنه يقوم بالنسخة من آخر مسافة موجودة في الجملة (*lastIndexOf*) إلى نهاية الجملة mySentence.length

والكود الكامل لهذه الفئة هو:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package newclass;

/*
 *
 * @author motaz
 */
public class Sentence {

    String mySentence;

    public Sentence (String atext){

       super();
       mySentence = atext;
    }

    public String getFirst(){

        String first;
        int firstSpaceIndex;
        firstSpaceIndex = mySentence.indexOf(" ");

        if (firstSpaceIndex == -1){
            first = mySentence;
        }
        else {
            first = mySentence.substring(0, firstSpaceIndex);
        }

        return (first);
    }

    public String getLast(){

        String last;
        int lastSpaceIndex;
        lastSpaceIndex = mySentence.lastIndexOf(" ");

        if (lastSpaceIndex == -1){
            last = mySentence;
        }
        else {
            last = mySentence.substring(lastSpaceIndex + 1, mySentence.length());
```

```
      }

        return (last);
    }
}
```

في كود الفورم الرئيسي للبرنامج MainForm.java قمنا بتعريف وتهيئة ثم استخدام هذا الكائن، واستقبلنا الجملة في مربع النص Text Field. وهذا هو الكود الذي يتم تنفيذه عند الضغط على الزر:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

        Sentence mySent = null;
        mySent = new Sentence(jTextField1.getText());

        jTextArea1.append("First: " + mySent.getFirst() + "\n");
        jTextArea1.append("Last: " + mySent.getLast() + "\n");
    }
```

قمنا بإنشاء كائن جديد وتهيئته في هذا السطر:

```
        mySent = new Sentence(jTextField1.getText());
```

والجُملة المُدخلة أثناء التهيئة تحصلنا عليها من مربع النص jTextField1 بواسطة الإجراء *getText* الموجود في هذا الكائن.

# المتغيرات والإجراءات الساكنة (static)

فيما سبق لنا من تعامل مع الفئات وجدنا أنه لابد من تعريف كائن من نوع الفئة قبل التعامل معها، فمثلاً لا نستطيع الوصول لإجراء الفئة بدون أن تصبح كائن. فنجد أن المثال التالي غير صحيح:

```
jTextArea1.append("First: " + Sentence.getFirst() + "\n");
```

لكن يُمكن استخدام إجراءات في فئات دون تعريف كائنات منها بتحويلها إلى إجراءات ساكنة static methods.

وهذا مثال لطريقة تعريف متغيرات وإجراءات ساكنة في لغة جافا:

```
public static class MyClass {
    public static int x;
    public static int getX(){
        return x;
    }
```

ويُمكن مناداتها مباشرة باستخدام إسم الفئة بدون تعريف كائن منها:

```
     MyClass.x = 10;
     System.out.println(MyClass.getX());
```

وبهذه الطريقة يُمكن أن يكون المتغير *x* مشتركاً في القيمة بين الكائنات المختلفة. لكن يجب الحذر والتقليل من استخدام متغيرات مشتركة Global variables حيث يصعب تتبع قيمتها ويصعب معرفة القيمة الحالية لها عند مراجعة الكود. والأفضل من ذلك هو استخدام إجراءات ثابتة يتم إرسال المتغيرات لها في شكل مُدخلات كما في المثال التالي والذي هو إجراء لتحويل الأحرف الأولى من الكلمات في جملة باللغة اللاتينية إلى حرف كبير Capital letter. وقد قُمنا بتسميتها هذه الفئة *Cap*:

```
public class Cap {

    public static String Capitalize(String input) {

        input = input.toLowerCase();
        char[] chars = input.toCharArray();

        for (int i=0; i < chars.length; i++) {
          if (i==0 || chars[i -1] == ' ')  {

            chars[i] = Character.toUpperCase(chars[i]);

        }
    }
    String result = new String(chars);
    return(result);
}
```

نلاحظ أننا قُمنا بكتابة إجراء من النوع الساكن static اسميناه *Capitalize* يقوم باستقبال متغير مقطعي اسمه *input* حيث يقوم بإرجاع متغير مقطي بعد تحويل بداية أحرفه إلى أحرف لاتينية كبيرة.
في البداية يتم تحويل كافة الجملة إلى حروف لاتنية صغيرة، ثم يتم نسخها إلى مصفوفة من نوع الرموز char ثم يتم تحويل الأحرف التي تلى المسافة إلى حروف كبيرة ويتم كذلك تحويل الحرف الأول في الجملة إلى حرف كبير. وفي النهاية تم نسخ تلك المصفوفة إلى متغير مقطعي جديد اسمه result ليتم إرجاعه في نداء الإجراء.
ويمكن مناداته مباشرة عن طريق إسم الفئة Cap بالطريقة التالية:

```
    String name = "motaz abdel azeem eltahir";
    System.out.println(Cap.Capitalize(name));
```

فتكون النتيجة كالتالي بعد التنفيذ:

```
Motaz Abdel Azeem Eltahir
```

يُمكن الإستفادة من الإجراءات الساكنة لكتابة مكتبة إجراءات مساعِدة عامة يُمكن استخدامها في عدد من البرامج. مثل إجراء لكتابة الأخطاء التي تحدث في ملف نصي والمعروف بالـ log file. أو تحويل التاريخ إلى شكل معين يُستخدم في نوعية معينة من البرامج، او غيرها من الإجراءات التي تُستخدم بكثرة لتوفير وقت للمبرمج.

## قاعدة البيانات SQLite

قاعدة البيانات SQLite هي عبارة عن قاعدة بيانات في شكل مكتبة معتمدة على ذاتها *self-contained* للتعامل مع قاعدة SQLite. ويُمكن استخدام طريقة SQL للتعامل معها. ويمكن استخدامها في أنظمة التشغيل المختلفة بالإضافة إلى الموبايل، مثلاً في نظام أندرويد أو BlackBerry

يمكن الحصول على المكتبة الخاصة بها وبرنامج لإنشاء قواعد بيانات SQLite والتعامل مع بياناتها من هذا الرابط:

http://sqlite.org/download.html

لإستخدامها في نظام وندوز نبحث عن ملف يبدأ بالإسم sqlite-shell، أما في نظام لينكس يمكننا تثبيت تلك المكتبة وأدواتها بواسطة مثبت الحزم. فقط نبحث عن الحزمة sqlite3

بعد ذلك نقوم بالإنتقال إلى شاشة الطرفية terminal لتشغيل البرنامج وهو من نوع برامج سطر الأوامر، ثم نختار دليل معين لإنشاء قاعدة البيانات ثم نكتب هذا الأمر:

```
sqlite3 library.db
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

بهذه الطريقة نكون قد أنشأنا قاعدة بيانات في ملف إسمه library.db

والآن مازلنا نستخدم هذه الأداة للتعامل مع قاعدة البيانات. ثم قُمنا بإضافة جدول جديد اسمه books بهذه الطريقة:

```
sqlite> create table books(BookId int, BookName varchar(100));
```

ثم أضفنا كتابين في هذا الجدول:

```
sqlite> insert into books values (1, "Introduction to Java 7");
sqlite> insert into books values (2, "One day trip with Java");
```

ثم عرضنا محتويات الجدول:

```
sqlite> select * from books;
1|Introduction to Java 7
2|One day trip with Java
sqlite>
```

الآن لدينا قاعدة بينات اسمها library.db وبها جدول اسمه books. يمكن الآن التعامل معها في برنامج جافا كما في المثال التالي.

# برنامج لقراءة قاعدة بيانات SQLite

قبل بداية كتابة اي برنامج لقاعدة بيانات SQLite بواسطة جافا يجب أن نبحث عن مكتبة جافا الخاصة بها. وهي مكتبة إضافية غير موجودة في آلة جافا الإفتراضية. ويُمكن الحصول عليها من هذا الموقع:

http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC

وإسم المكتبة هو sqlite-jdbc ثم نختار رقم النسخة المناسب. في هذه الأمثلة اخترت الملف:

```
sqlite-jdbc-3.7.2.jar
```

من هذا الرابط:

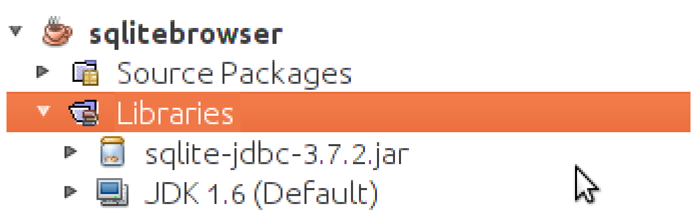
https://bitbucket.org/xerial/sqlite-jdbc/downloads

يُفضل اختيار النسخة الأحدث دائماً.

وهذه المكتبة هي كُل مانحتاجه للتعامل مع قاعدة البيانات SQLite في برامج جافا، فهي لاتحتاج لمخدم لتثبيته حتى تعمل قاعدة البيانات كما قُلنا سابقاً.

قمنا بفتح مشروع جديد أسميناه sqlitebrowser لعرض قاعدة البيانات Library التي قُمنا بإنشائها سابقاً.

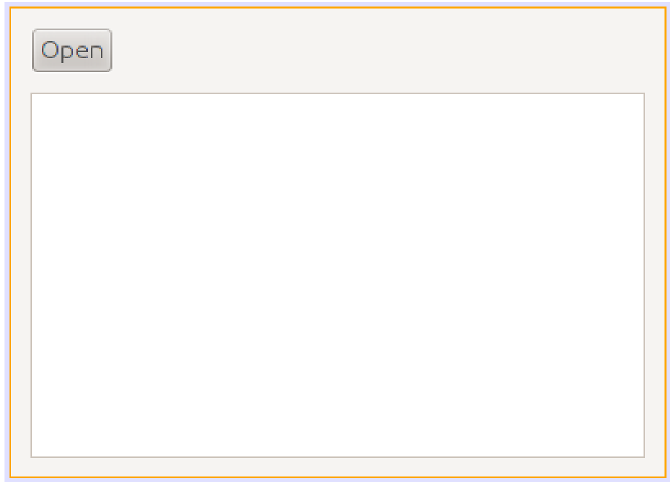
في شاشة المشروع يوجد فرع أسمه Libraries نقف عليه ثم نختار بالزر اليمين للماوس Add JAR/Folder ثم نختار الملف sqlite-jdbc-3.7.2.jar الذي قُمنا بتحميله من الإنترنت سابقاً.




أضفنا JFrame Form وأسميناه MainForm واستدعيناه من الملف الرئيسي Sqlitebrowser.java بالطريقة التالية:

```
    public static void main(String[] args) {
        MainForm form = new MainForm();
        form.setVisible(true);
    }
```

في الفورم أضفنا زر و مربع نص TextArea بالشكل التالي:




ثم أضفنا فئة كائن جديد New class أسميناه SqliteClient وهو الكائن الذي سوف يحتوي على إجراءات قراءة قاعدة بيانات *SQLite* والكتابة فيها.

قمنا بتعريف الكائن *dbConnection* من نوع *Connection* داخل كود فئة الكائن *SqliteClient* لتعريف مسار قاعدة البيانات والإتصال بها للإستخدام لاحقاً في باقي إجراءات الكائن SqliteClient.

ثم قمنا بكتابة الكود لإستقبال إسم قاعدة البيانات ثم الإتصال بها في الإجراء الرئيسي *constructor* لهذا الكائن:

```
public class SqliteClient {
        Connection dbConnection = null;

// Constructor
public SqliteClient (String aDatabaseName) {
        super();
        try {
          Class.forName("org.sqlite.JDBC");
          dbConnection = DriverManager.getConnection(
            "jdbc:sqlite:" + aDatabaseName);
        }
        catch (Exception e){
            System.out.println("Error while connecting: " + e.toString());
        }

    }
```

نلاحظ أننا قُمنا بحماية الكود بواسطة *try .. catch* وذلك لأنه من المتوقع أن تحدث مشكلة أثناء التشغيل، مثلاً أن تكون قاعدة البيانات المُدخلة غير موجودة، أو أن مكتبة SQLite غير موجودة.

الإجراء الأول(Class.forName) يقوم بتحميل مكتبة SQLite لنتمكن من نداء الإجراءات الخاصة بهذه القاعدة من تلك المكتبة التي قُمنا بتحميلها من الإنترنت، فإذا لم تكن موجودة سوف يحدث خطأ.

في السطر التالي قمنا بتهيئة الكائن dbConnection وإعطائه إسم الملف التي تم إرساله عند تهيئة الكائن SqliteClient

بعد ذلك قُمنا بإضافة الإجراء *showTable* إلى فئة الكائن SqliteClient لعرض محتويات الجدول المرسل لهذا الإجراء في مربع النص:

```
public boolean showTable(String aTable, JTextArea textArea) {

    ResultSet myRecords = null;
    Statement myQuery = null;

    try {
      myQuery = dbConnection.createStatement();
      myRecords = myQuery.executeQuery("SELECT * from " + aTable);

      // Read records
      while (myRecords.next())
      {
        textArea.append(myRecords.getString(1) + " - "
                 + myRecords.getString(2) + "\n");
      }

     catch (Exception e){

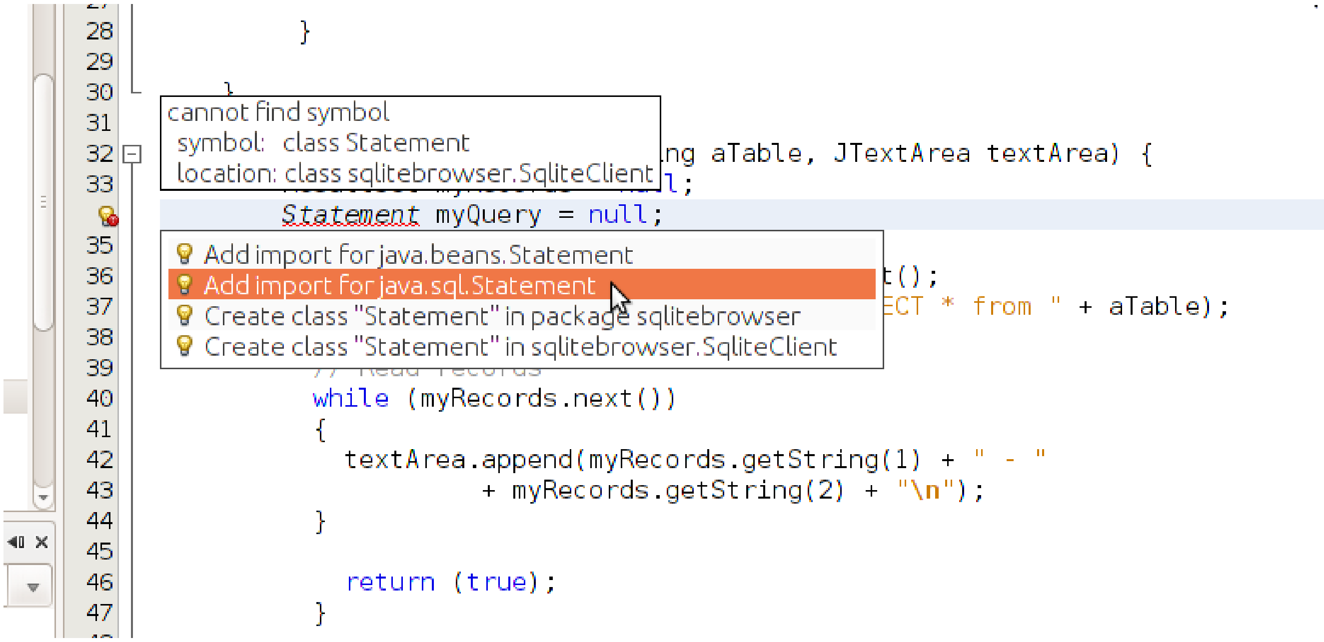
        textArea.append("Error while reading table: " + e.toString() + "\n");
        return (false);
    }
}
```

قمنا في هذا الإجراء بتعريف كائن من نوع *Statement* أسميناه *myQuery* يسمح لنا بكتابة query بلغة SQL على

قاعدة البيانات.

وعند إضافة المكتبة المحتوية على الكائن *Statement* لابد من أن ننتبه لإختيار المكتبة java.sql.Statement ولا نختار الخيار الأول الذي يظهر عند الإضافة التلقائية java.beans.Statement التي تتسبب في أخطاء أثناء الترجمة.

الخيار الصحيح للمكتبة يظهر في الشكل التالي:




ثم قمنا بتهيئته على النحو التالي:

```
        myQuery = dbConnection.createStatement();
```

ثم قمنا بنداء الإجراء executeQuery في الكائن myQuery وأعطيناه مقطع SQL والذي به أمر عرض محتويات الجدول. هذا الإجراء يُرجع كائن جديد من هو عبارة عن حزمة البيانات ResultSet. استقبلناه في الكائن *myRecords* والذي هو من نوع فئة الكائن ResultSet والذي قُمنا بتعريفه في بداية الإجراء دون تهيئته.

بعد هذه الإجراءات قُمنا بالمرور على كل السجلات في هذا الجدول وعرضنا بعض الحقول  في مربع النص الذي تم إرساله كُمدخل للإجراء *showTable*:

```
      // Read records
      while (myRecords.next())
      {
        textArea.append(myRecords.getString(1) + " - "
                  + myRecords.getString(2) + "\n");
      }
```

والإجراء *next* يقوم بتحريك مؤشر القراءة لبداية الجدول أو حزمة البيانات ثم الإنتقال في كل مرة إلى السجل الذي يليه ويرجع القيمة true. وعندما تنتهي السجلات أو لايكون هُناك سجلات من البداية ترجع القيمة false وعندها تتوقف الحلقة.

داخل الحلقة قرأنا الحقل الأول والثاني من الجدول المُرسل بواسطة الدالة *getString* وأعطيناها رقم الحقل Field/Column وهي تُرجع البيانات في شكل مقطع، ويُمكن استخدامها حتى مع الأنواع الأخرى مثل الأعداد الصحيحة مثلاً أو التاريخ، فكلها يُمكن تمثيلها في شكل مقاطع.

في الإجراء التابع للزر *Open* في الفورم الرئيسي MainForm قمنا بكتابة هذا الكود لعرض سجلات الجدول books الموجود في قاعدة البيانات library.db

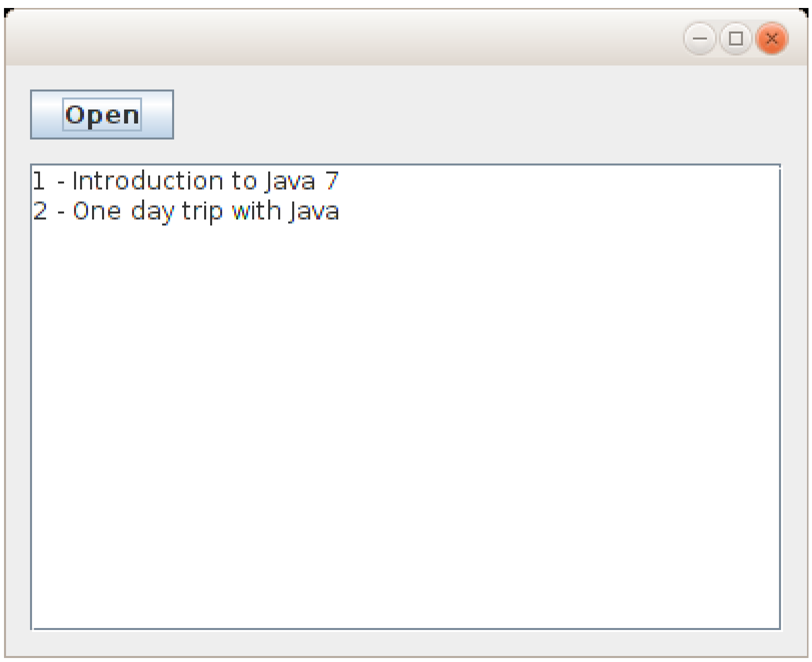
```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

        SqliteClient sql = new SqliteClient("/home/motaz/library.db");

        jTextArea1.setText("");
        sql.showTable("books", jTextArea1);
    }
```

في السطر الأول قمنا بتعريف الكائن *sql* من نوع الفئة التي قمنا بإنشائها SqliteClient ثم تهيئتها بإرسال إسم ملف قاعدة البيانات. وهذا المثال لبرنامج في بيئة لينكس.

ثم في السطر الثاني قمنا بحذف محتويات مربع النص. ثم في السطر الثالث إستدعينا الإجراء showTable في هذا الكائن لعرض محتويات الجدول books وكانت النتيجة كالتالي:




لإضافة كتاب جديد في قاعدة البيانات في الجدول tables أولاً نقوم بإضافة إجراء جديد نسميه مثلاً *insertBook* في فئة الكائن SqliteClient بعد الإجراء showTable. وهذا هو الكود الذي كتبناه لإضافة كتاب جديد:

```
  public boolean insertBook(int bookID, String bookName) {

      try {
```

```
            PreparedStatement insertRecord = dbConnection.prepareStatement(
                 "insert into books (BookID, BookName) values (?, ?)");

           insertRecord.setInt(1, bookID);
           insertRecord.setString(2, bookName);
           insertRecord.execute();
           return(true);
         }

         catch (Exception e){
           System.out.println("Error while reading table: " + e.toString());
           return (false);
       }
```

في العبارة الأولى لهذا الإجراء قمنا بتعريف الكائن *insertRecord* من نوع الفئة PreparedStatement وهي تُستخدم لتنفيذ إجراء على البيانات DML مثل إضافة سجل، حذف سجل أو تعديل. ونلاحظ أننا وضعنا علامة إستفهام في مكان القيم التي نُريد إضافتها في الجزء values. وهذه تُسمى مدخلات parameters. سوف يتم تعبئتها لاحقاً.

في العبارة الثانية وضعنا رقم الكتاب في المُدخل الأول بواسطة *setInt*، ثم في العبارة الثالثة وضعنا إسم الكتاب في المُدخل الثاني بواسطة *setString*، ثم قمنا بتنفيذ هذا الإجراء بواسطة execute والتي تقوم بإرسال طلب الإضافة هذا إلى مكتبة SQLite والتي بدورها تقوم بتنفيذه في ملف قاعدة البيانات library.db

بعد ذلك نضيف فورم ثاني من نوع JFrame Form ونسميه AddForm نضع فيه المكونات jLabel و jTextField بالشكل التالي:




ولاننسى تحويل خاصية إغلاق الفورم defaultCloseOperation إلى Dispose بدلاً من Exit_On_Close حتى يتم

إغلاق البرنامج عند إغلاق هذا الفورم الغير رئيسي.

وفي حدث الزر *Insert* نكتب فيه الكود التالي:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

        SqliteClient query = new SqliteClient("/home/motaz/library.db");


        int bookID;
        bookID = Integer.parseInt(jTextField1.getText().trim());
        query.insertBook(bookID, jTextField2.getText());
        setVisible(false);
    }
```

في السطر الأول قمنا بتعريف الكائن query من نوع الفئة SqliteClient والتي تحتوي إجراء الإضافة الذي أضفناه

مؤخرا.

في السطر الثاني قمنا بتعريف المتغير bookID من نوع العدد الصحيح.

الحقلjTextField1 يُرجع محتوياته بواسطة الإجراء getText في شكل مقطع String. ونحن نريده أن يستقبل رقم

الكتاب وهو من النوع الصحيح والمقطع يمكن أن يحتوي على عدد صحيح. فقمنا بتحويل المقطع إلى عدد صحيح

بعد حذف أي مسافة غير مرغوب فيها -إن وجدت- بواسطة الدالةtrim الموجودة في الكائن String، وذلك لأن العدد إذا كان يحتوي على حروف أو رموزأخرى أو مسافة فإن التحويل إلى رقم بواسطة الإجراءparseInt سوف ينتج عنها خطأ. والدالة trim تقوم بإرجاع مقطع محذوفة فيه المسافة من بداية ونهاية النص، لكنها لاتؤثر على الكائن الذي تم تنفيذها فيه. مثلاً الكائن jTextField1 لايتم حذف المسافة منه. لتوضيح ذلك انظر المثال التالي:

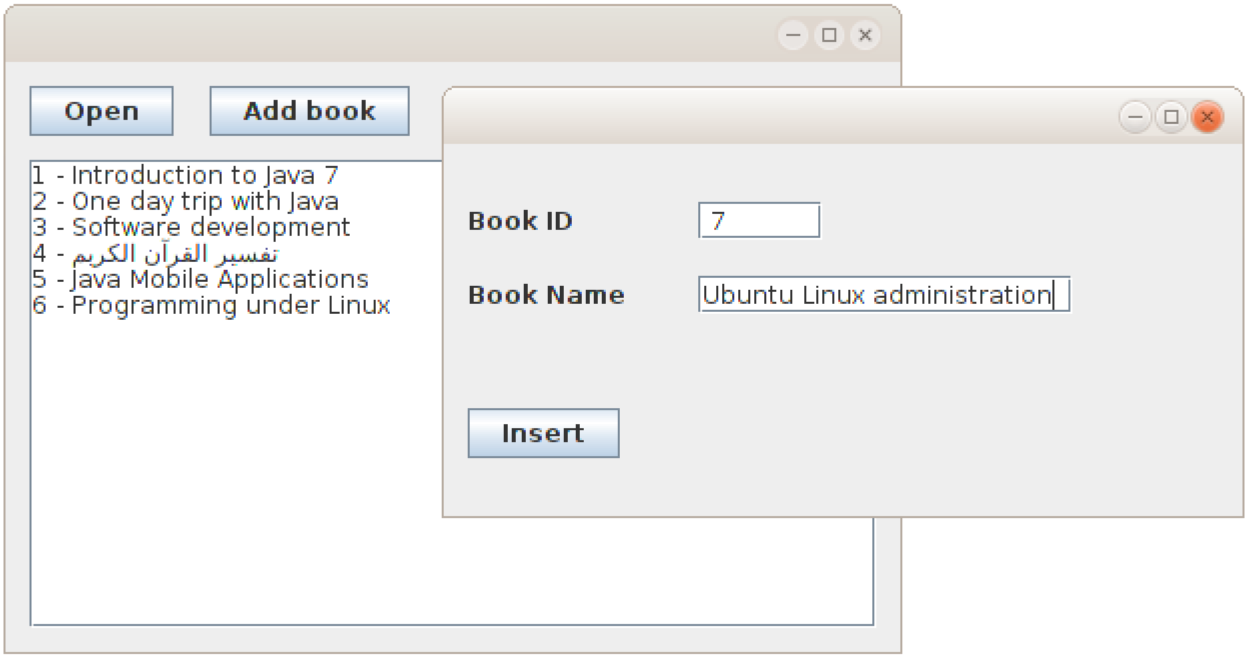
```
        myText = aText.trim();
```

الكائن aText لايتثأر بالدالة trim أما الكائن myText فتيم تخزين مقطع فيه من المقطع aText بدون مسافة.

في السطر الرابع قمنا بإستدعاء الإجراء insertBook في الكائن query وأعطيناه رقم الكتاب في المدخل الأول ثم

اسم الكتاب في المدخل الثاني. ثم قمنا بإغلاق الفورم في السطر الأخير بواسطة setVisible وأعطيناها القيمة false.

وهذا هو شكل البرنامج بعد تنفيذه:




عند عمل build لهذا البرنامج بواسطة shift + F11 نلاحظ وجود دليل فرعي إسمه lib داخل الدليل distوهو يحتوي على المكتبة التي استخدمناها والتي هي ليست جزء من آلة جافا الإفتراضية. وعند نقل البرنامج إلى أجهزة أخرى لابد

من نقل الدليل lib مع البرنامج، وإلا تعذر تشغيل إجراءات قاعدة البيانات.

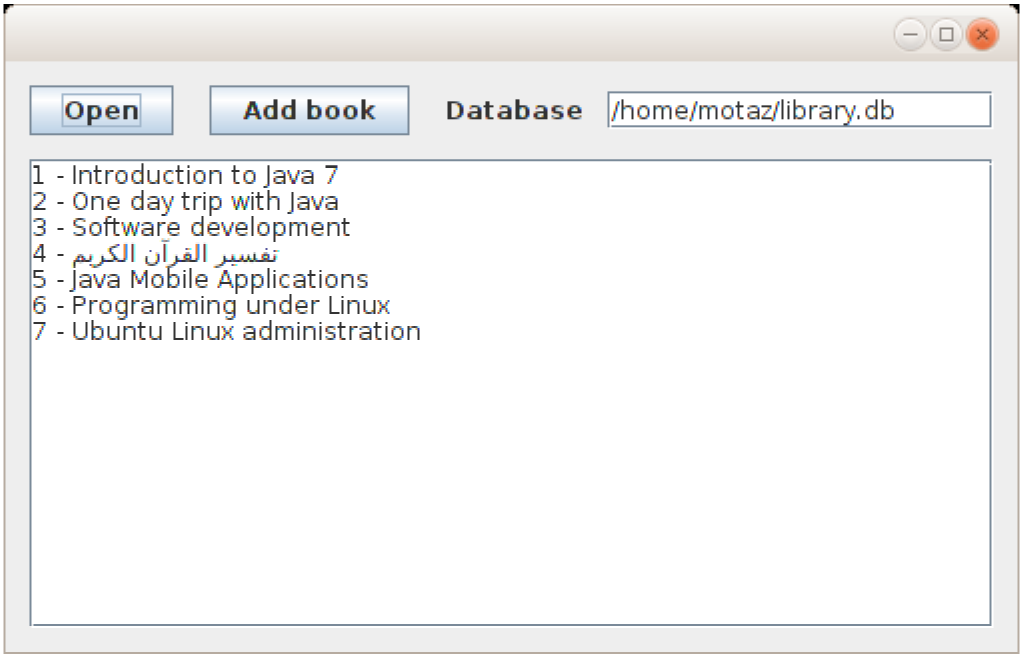
في هذا المثال نحتاج لنقل ثلاث ملفات:

- `sqlitebrowser.jar` وهو الملف التنفيذي للبرنامج في صيغة Byte code
- `sqlite-jdbc-3.7.2.jar` وهو ملف المكتبة داخل الدليل *lib* .

ولابد أن نضعه في هذا الدليل داخل الدليل الذي نضع فيه البرنامج

- `library.db` وهو ملف قاعدة البيانات. وكان من الأفضل جعل مسار قاعدة البيانات خارج كود البرنامج، مثلاً

نضيف jTexField آخر ليكون البرنامج بالشكل التالي:




ونغير تهيئة كائن قاعدة البيانات بالشكل التالي:

```
        SqliteClient sql = new SqliteClient(jTextField1.getText());
```

بهذه الطريقة يكون البرنامج أكثر حرية في النقل portable ولايعتمد على ثوابت في نظام معين. وهي طريقة جيدة

في تطوير البرامج تزيد من إمكانية إستخدامه، خصوصا عند إختيار لغة برمجة متعددة المنصات مثل جافا.

# تكرار حدث بواسطة مؤقت

في هذا المثال نُريد كتابة التاريخ والساعة في الشاشة كل فترة معينة، مثلاً كل ثانية. ولعمل ذلك قمنا بفتح مشروع جديد سميناه timer ثم أضفنا MainForm من نوع JFrame Form، ثم أضفنا فئة جديدة new class أسميناها MyTimer فكان تعريفها بالشكل التالي:

```
public class MyTimer {
```

لكن قمنا بتغير هذا التعريف لنستخدم مايُعرف بالوراثة inheritance وذلك بدلاً من كتابة فئة كائن جديد من الصفر نستخدم كائن لديه خصائص مشابهة ثم نزيد فيها. وفئة هذا الكائن اسمها TimerTask. نقوم بوراثته بهذه الطريقة:

```
public class MyTimer extends TimerTask{
```

ثم نقوم بتعريف كائن myLabel بداخله حتى نقوم بعرض التاريخ والوقت فيه، ثم قمنا بكتابة إجراء التهيئة:

```
    JLabel myLabel;

    public MyTimer(JLabel alabel){
        super();
        myLabel = alabel;
    }
```

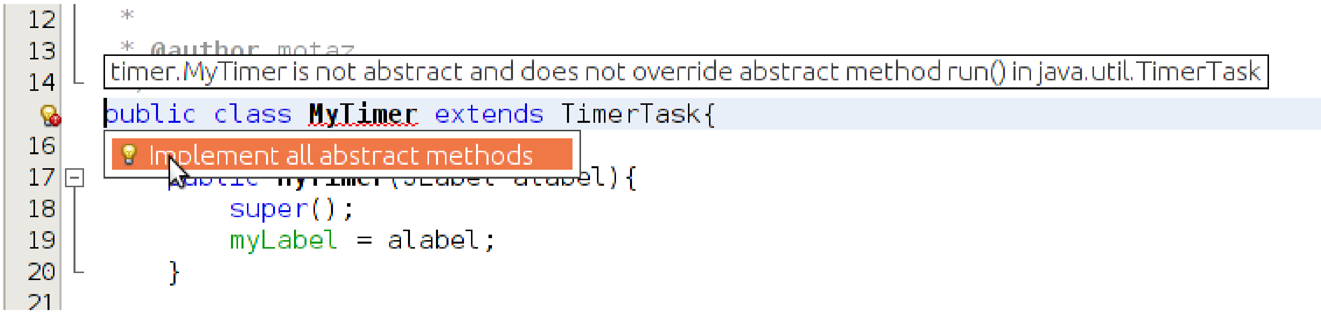
وفي هذا الإجراء نستقبل الكائن alabel ثم نقوم بحفظ نسخة منه في الكائن myLabel.

بعد ذلك نقوم بكتابة الإجراء الذي سوف يتم إستدعائه كل فترة وأسمه run بهذه الطريقة:

```
    @Override
    public void run() {
        Date today = new Date();
        myLabel.setText(today.toString());

    }
```

ويمكن إضافة تعريف هذا الإجراء تلقائياً بواسطة implement all abstract methods والتي تظهر في سطر تعريف الكائن MyTimer بالطريقة التالية:




وهذا هو كود الكائن كاملاً:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package timer;

import java.util.Date;
import java.util.TimerTask;
import javax.swing.JLabel;

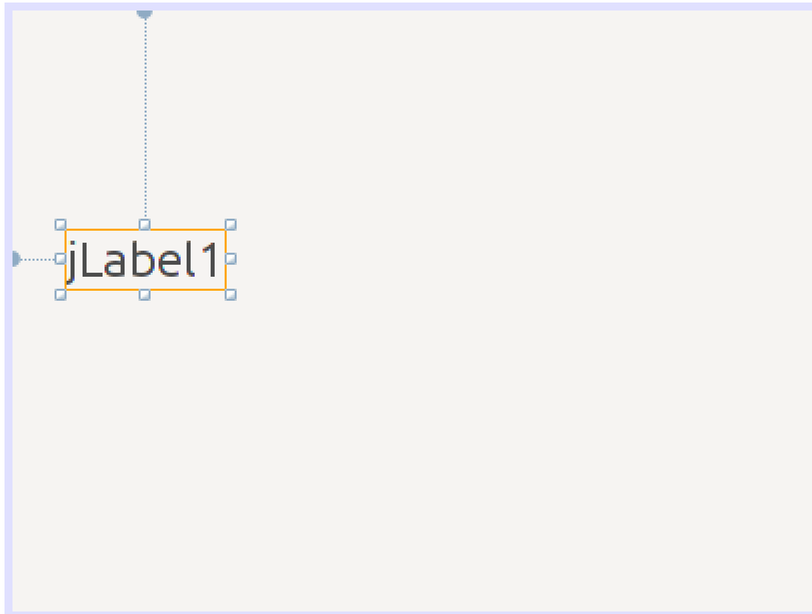
/*
 *
 * @author motaz
 */
public class MyTimer extends TimerTask{
    JLabel myLabel;
    public MyTimer(JLabel alabel){
        super();
        myLabel = alabel;
    }

    @Override
    public void run() {
        Date today = new Date();
        myLabel.setText(today.toString());

    }


}
```

نضع jLabel في الفورم الرئيسي MainForm ونزيد حجم الخط فيه ليكون بالشكل التالي:




في إجراء تهيئة هذا الفورم نعدل الكود إلى التالي:

```
    public MainForm() {
        initComponents();
        java.util.Timer generalTimer = null;

        MyTimer timerObj = new MyTimer(jLabel1);
        generalTimer = new java.util.Timer("time loop");
        generalTimer.schedule(timerObj, 2000, 1000);
    }
```

في هذا السطر:

```
        java.util.Timer generalTimer = null;
```

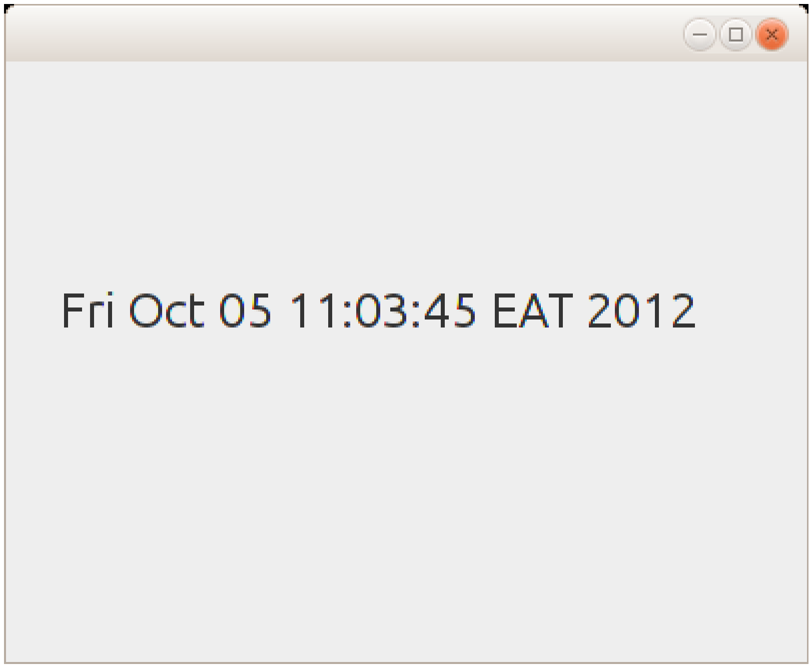
قمنا بتعريف الكائن generalTimer من النوع Timer. وهذا الكائن لديه خاصية تكرار الحدث بفترة زمنية محددة.

ثم قمنا بتعريف الكائن timerObj من الفئة MyTimer والتي قمنا بكتابتها لإظهار التاريخ والوقت.

ثم قمنا بتهيئة الكائن generalTimer. وفي السطر الأخير قمنا بتشغيل المُؤقت shedule وأرسلنا له الكائن timerObj ليقوم بتنفيذ الإجراء run كل فترة معينة. والرقم الأول 2000 هو بداية التشغيل أول مرة، وهو بالملي ثانية، أي يقوم بالإنتظار ثانيتين ثم التشغيل أول مرة.

الرقم الثاني 1000 هو التكرار بالملي ثانية أيضاً. حيث يقوم بإظهار التاريخ والوقت كل ثانية.

نقوم بتشغيل البرنامج لنرى أن الثواني تتغير في الفورم الرئيسي:



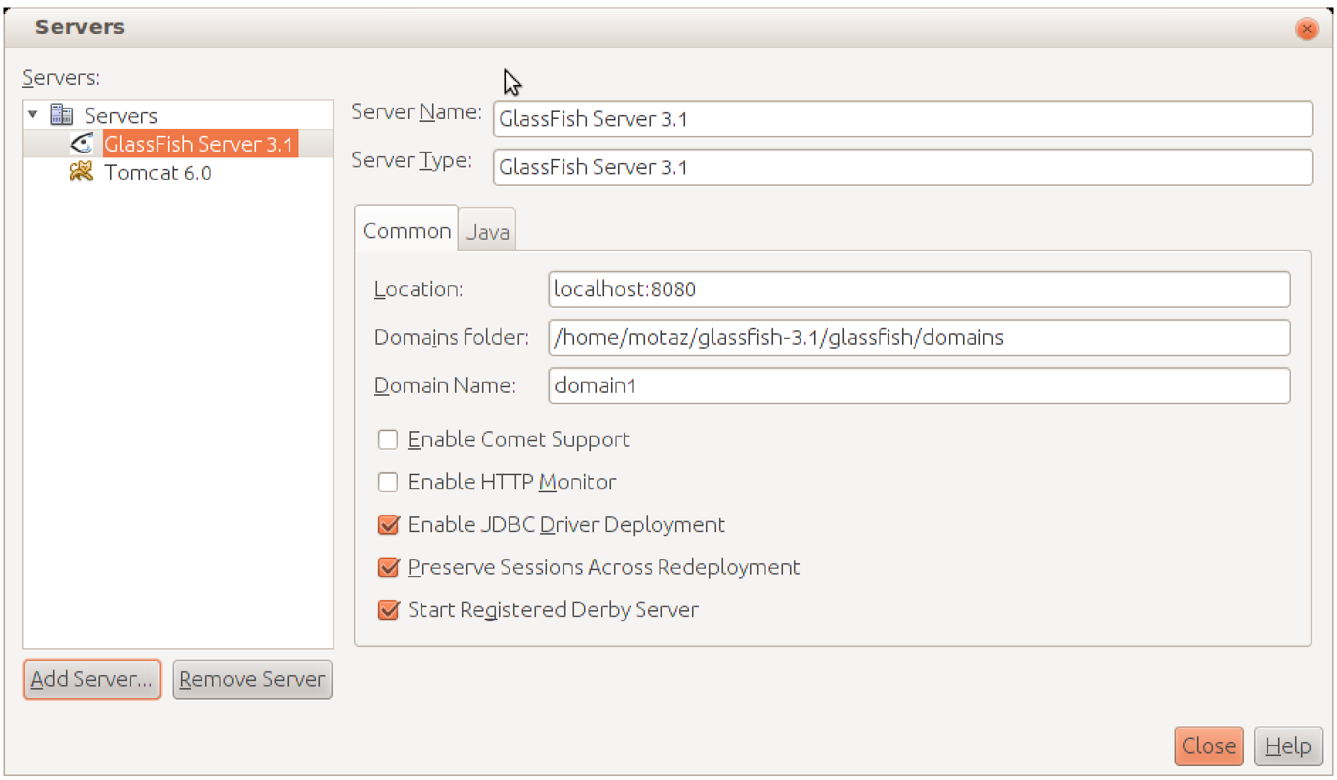

وسوف يتم تشغيل هذا الإجراء تلقائياً إلى إغلاق البرنامج.

# برمجة الويب بإستخدام جافا

تعتبر لغة جافا من أهم اللغات التي تدعم برمجة الويب web applications وخدمات الويب web services مثل الـ SOAP والـ RESTfull. والتقنية التي استخدمناها هنا في شرح برامج الويب هي تقنية Servlet . وفي النهاية يتم تشغيل برامج الويب وخدمات الويب المكتوبة بجافا في مخدم ويب خاص هو Apache Tomcat أو GlassFish.

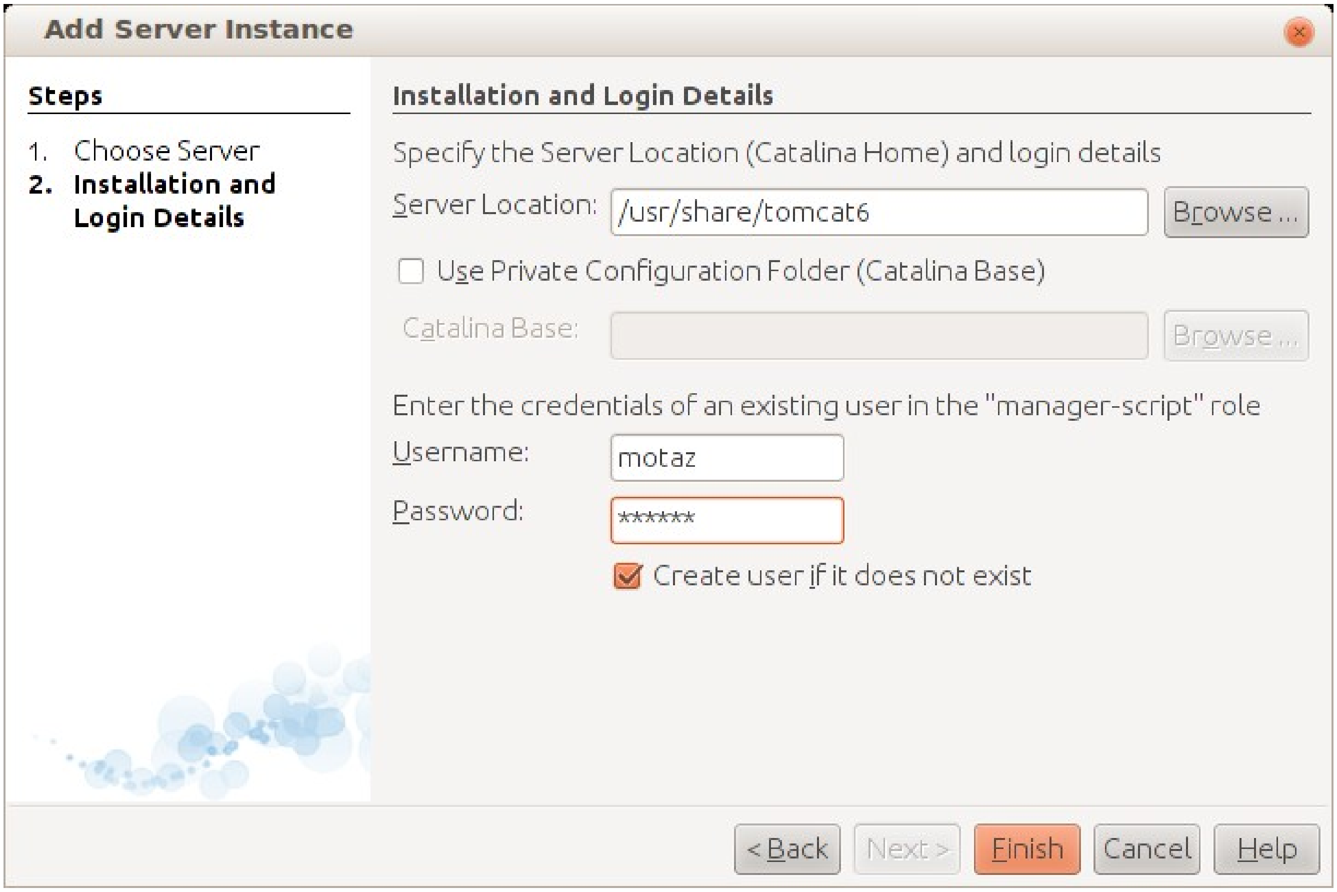
# تثبيت مخدم الويب

قبل البداية في كتابة برامج ويب يجب التأكد من أنه يوجد مخدم ويب خاص بجافا وأن له إعدادات في بيئة التطوير NetBeans وذلك عن طريق Tools/Servers فتظهر هذه الشاشة:




ويظهر فيها وجود GlassFish و Tomcat 6.

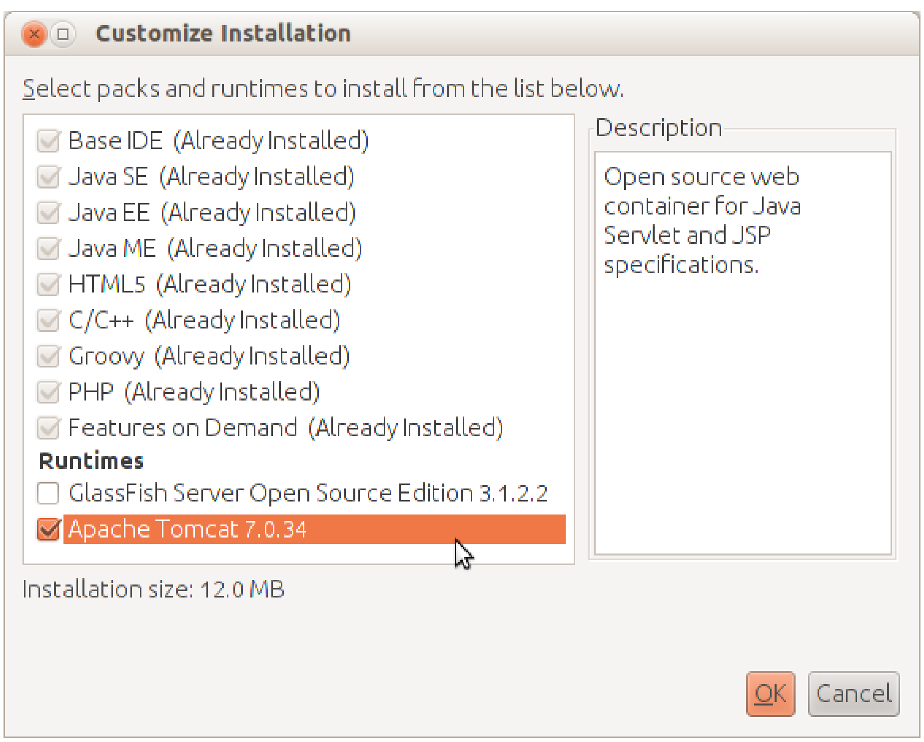
وفي حالة عدم وجود Tomcat مثلاً نقوم بالحصول عليه من موقع tomcat.apache.org أو عن طريق مدير الحزم في نظام لينكس. ثم نقوم بإضافة مخدم جديد عن طريق الزر Add Server ثم نختار رقم نسخة Tomcat التي قمنا بتثبيتها ثم نُدخل الدليل الذي نزلت فيه كما في هذا المثال:


Add Server Instance
Steps
1. Choose Server
2. Installation and Login Details
Installation and Login Details
Specify the Server Location (Catalina Home) and login details
Server Location:
/usr/share/tomcat6
Browse ...
Use Private Configuration Folder (Catalina Base)
Catalina Base:
Browse ...
Enter the credentials of an existing user in the "manager-script" role
Username:
motaz
Password:
******
Create user if it does not exist
< Back
Next >
Finish
Cancel
Help

يُمكن إختيار Tomcat أثناء تثبيت NetBeans وذلك بإختيار Customize كما تظهر في هذه الصورة:



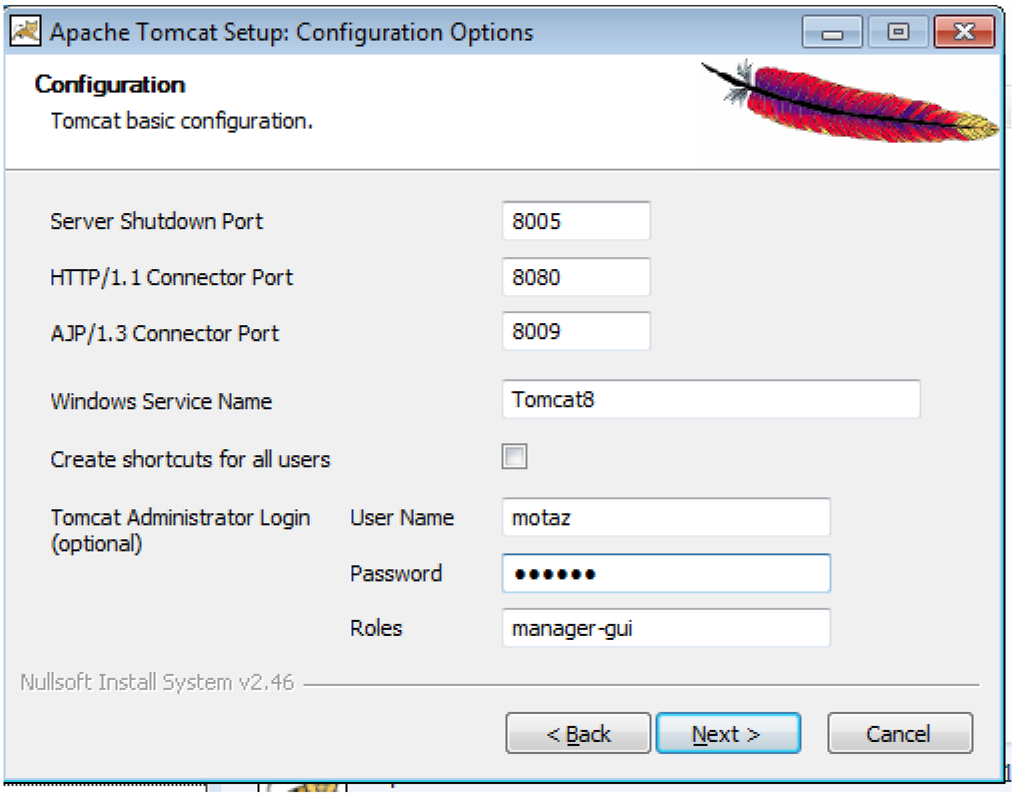

ثم اختيار Tomcat :



عند الحاجة لتثبيت برامج الويب في أجهزة أخرى (مخدم مثلاً) نقوم بتثبيت نسخة Apache Tomcat بطريقة منفصلة حسب نظام التشغيل المستهدف، من الموقع:

http://tomcat.apache.org/

ثم نقوم بتحميل الحزمة المناسبة ثم تثبيتها ثم بإدخال إسم دخول لديه صلاحية لتثبيت برامج الويب. في نظام وندوز يمكن إدخال الإسم أثناء تثبيت البرنامج:




أما في نظام لينكس فنبحث أولاً عن الملف tomcat-users.xml في دليل الإعدادات حسب نظام التشغيل، وحسب نسخة الـ tomcat مثلاً في بيئة لينكس يكون في هذا المسار:

```
/etc/tomcat6/tomcat-users.xml
```

في نظام وندوز يكون في هذا المسار، وذلك إذا احتجنا إلى تغيير اسم الدخول أو كلمة المرور:

```
C:\Program Files\Apache Software Foundation\Tomcat 8.0
```

نقوم بإضافة هذا السطر لمستخدم جديد أو تعديل مستخدم موجود لإعطائه الصلاحيات الكافية:

```
<user username="motaz" password="tomcat" roles="manager-gui,manager-script"/>
```

عندها تكون البيئة جاهزة لتثبيت وتشغيل برامج أو خدمات ويب.

# أول برنامج ويب

لعمل برنامج الويب الأول في جافا نقوم بعمل مشروع جديد عن طريق

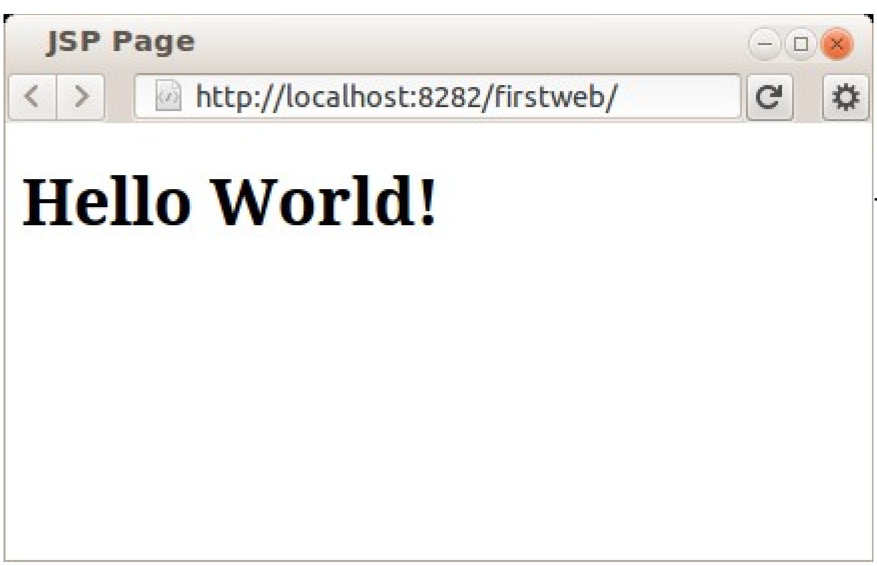
```
New project\Java Web\Web Application
```

ثم نختار إسم للمشروع مثلاً *firstweb* ثم نختار المخدم، وهو في هذه الحال Tomcat 6 ثم نضغط على زر Finish.

عندها يتم إضافة كود HTML تلقائياً كصفحة رئيسة كما يلي:

```
<%"page contentType="text/html" pageEncoding="UTF-8@%>
<DOCTYPE html!>
<html>
<head>
<"meta http-equiv="Content-Type" content="text/html; charset=UTF-8>
<title>JSP Page</title>
<head/>
<body>
<h1>Hello World!</h1>
<body/>
<html/>
```

يُمكن تشغيل البرنامج مباشرة من بيئة NetBeans بواسطة المفتاح F6 ليتم عرض هذه الصفحة بواسطة تشغيل مخدم Tomcat لتظهر على متصفح خاص بالشكل التالي:



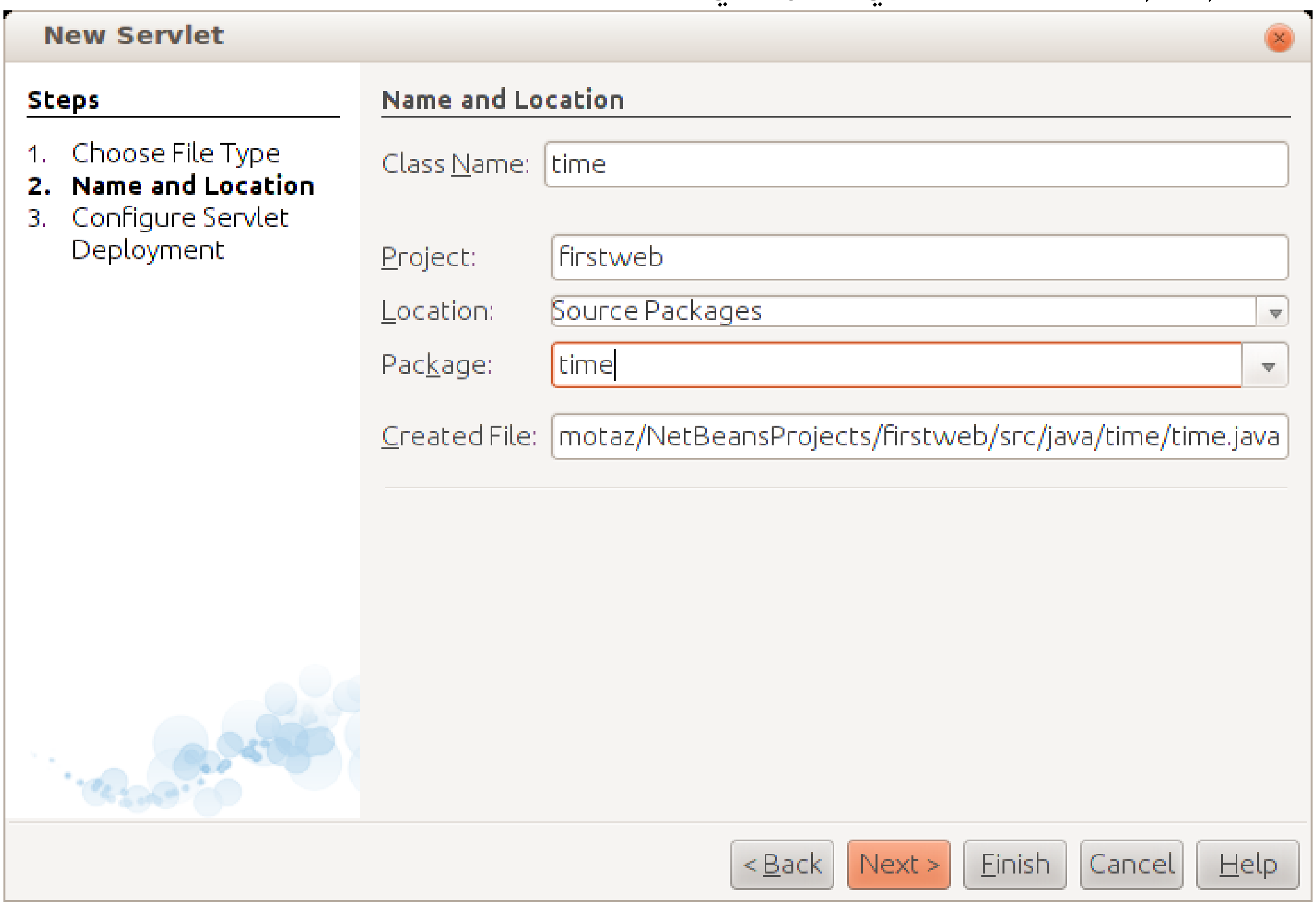
نلاحظ أن العنوان هو:

http://localhost:8282/firstweb/

ويظهر فيه رقم المنفذ 8282 والمنفذ الإفتراضي Default port هو 8080 وقمت بتغييره للتمكن من تشغيل GlassFish أيضاً. حيث يُمكن أن يكون في نفس الجهاز عدد من مخدمات الويب التي تعمل في نفس الوقت بشرط أن يكون لكل واحد رقم منفذ مختلف. مثلاً مخدم Apache يعمل في المنفذ الإفتراضي للويب وهو 80 وهو لايحتاج لكتابة في عنوان الموقع، و Tomcat في المنفذ 8282 و GlassFish في المنفذ 8080.

نقوم بإغلاق المتصفح لنرجع للبرنامج لنضيف فيه محتوي تفاعلي، حيث أن الصفحة السابقة كانت صفحة ثابتة static أو أنها تستخدم تقنية مختلفة وهي JSP.

في شجرة المشروع وفي الفرع Source Packages نقوم بإضافة Servlet عن طريق الزر اليمين للماوس ثم New. ثم نقوم بتسميته timer كما في الشكل التالي:




نقوم بتسمية الحزمة package بنفس الإسم مثلاً. ليظهر كود تلقائي يحتوي على الإجراء *ProcessRequest* كما في المثال التالي:

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
```

```
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            /* TODO output your page here
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet time</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet time at " + request.getContextPath () + "</h1>");
            out.println("</body>");
            out.println("</html>");
             */
        } finally {
            out.close();
        }
    }
```

نقوم بحذف الجزء المعطل commented ثم نقوم بضافة هذه الأسطر في داخل العبارة try

```
            Date today = new Date();
            out.println("Time in server is: <b>" + today.toString() + "</b>");
```

ثم نقوم بتشغيله مرة أخرى لكن نضيف كلمة timer في آخر العنوان في متصفح الويب كالتالي:



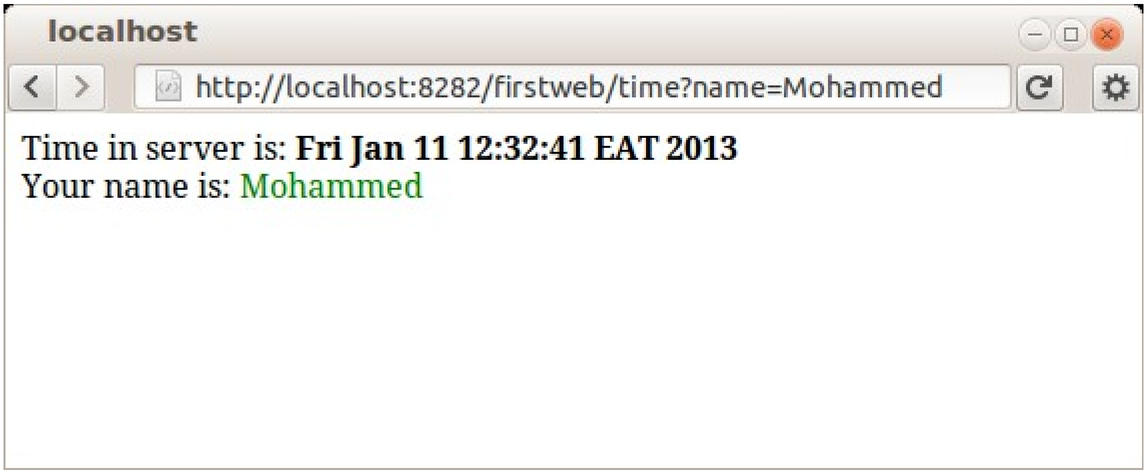

لإستقبال مُدخلات عن طريق العنوان نضيف السطر التالي في البرنامج:

```
out.println("<br>Your name is: <font color=green>" + request.getParameter("name") +
            "</font>");
```

وذلك بإعتبار أنه سوف يتم إدخال إسم المستخدم  في العنوان كالتالي:

http://localhost:8282/firstweb/time?name=Mohammed

فتكون النتيجة كالتالي في المتصفح:




# تثبيت برامج الويب

بعد الإنتهاء من تطوير برامج الويب نقوم بإنتاج نسخة تنفيذية بواسطة Clean and Build لنتحصل على الملف firstweb.war وهو ملف تنفيذي يُمكن وضعه في مخدم Tomcat الذي نُريد تثبيت البرنامج فيه. ونضعه في الدليل webapps ضمن دليل برنامج Tomcat. مثلاً في نظام لينكس يكون إسم الدليل هو:

```
/var/lib/tomcat6/webapps/
```

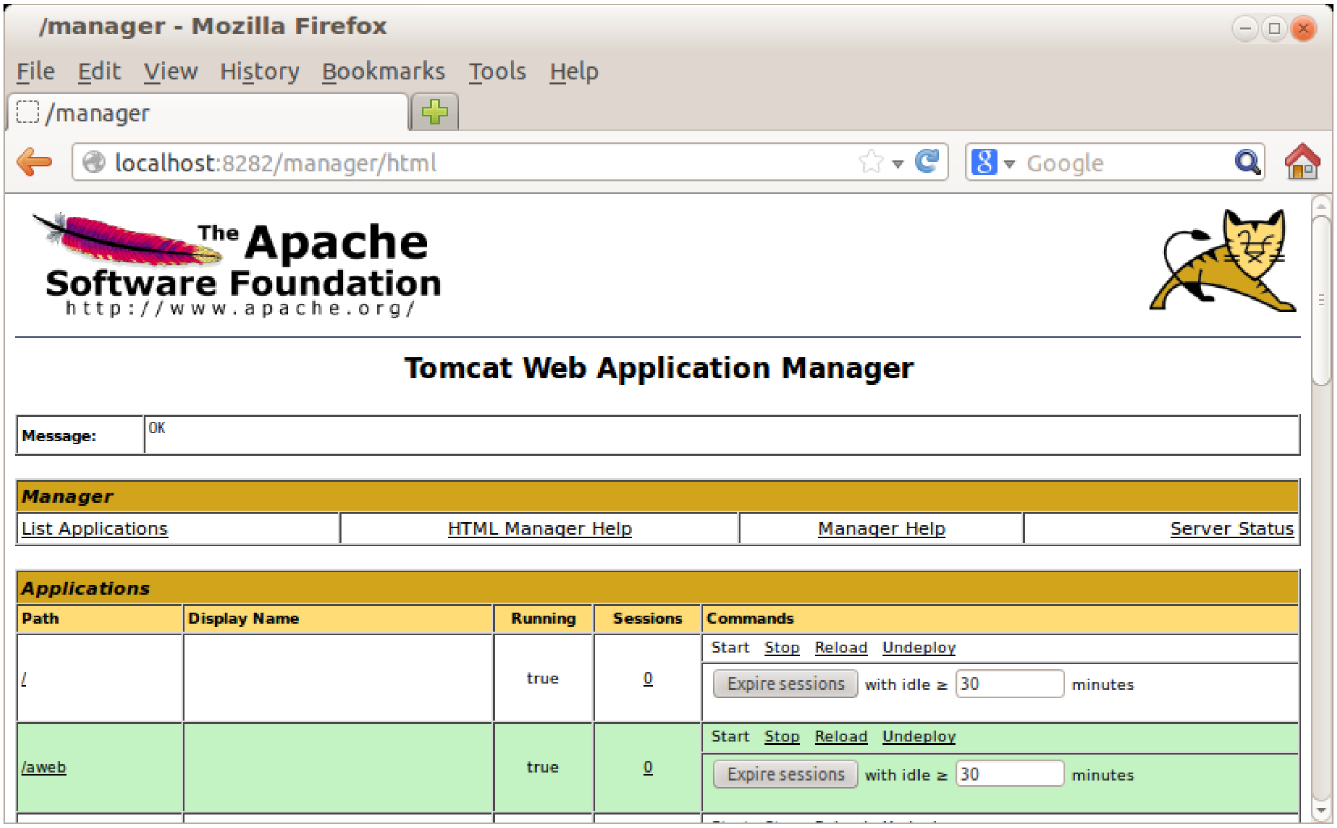
وفي نظام ويندوز نجده في الدليل :

```
Program Files\Apache Software foundation\Tomcat 6\webapps
```

لانحتاج لنسخ الملف يدوياً إلى الدليل webapps بل نستخدم مدير برامج الويب في مخدم Tomcat عن طريق المتصفح، حيث نكتب العنوان التالي:

```
http://localhost:8282
```

ثم نضغط رابط Manager Webapp لتظهر لنا الشاشة التالية في المتصفح:




في الجزء Deploy/War file to deploy

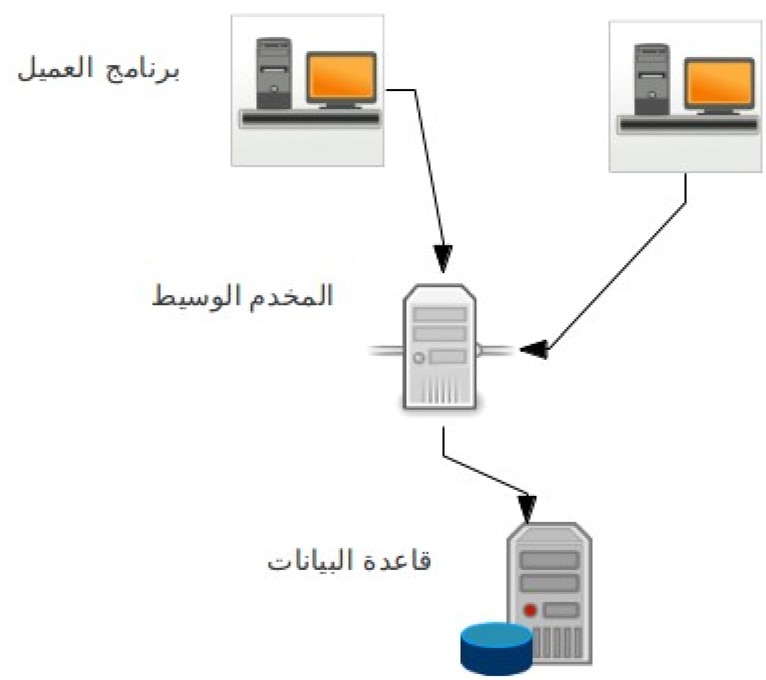
نقوم برفع الملف ثم نضغط على زر Deploy ليتم نسخ الملف في دليل webapps ليصبح متوفراً للإستخدام.

هذه كانت مقدمة فقط لبرمجة الويب بواسطة لغة جافا لإعطاء فكرة عامة وكبداية. يُمكن البحث أكثر عن هذا الموضوع لعمل برامج ويب متطورة.

# خدمات الويب Web services

خدمات الويب هي عبارة عن برامج مشابهة لبرامج الويب إلا أن الفرق الأساسي هو أن برامج الويب كما في المثال السابق تتم كتابتها ليتم الوصول إليها عن طريق المتصفح مباشرة، أما خدمات الويب فيتم الوصول إليها بواسطة برامج أخرى. وبمعنى آخر في برامج الويب يقوم المتصفح بطلب عنوان صفحة أو action فيتم الرد على المتصفح بتحميل صفحة HTML أو صورة أو غيرها من الأشياء التي يستطيع التعامل معها متصفح الويب. أما خدمة الويب فيتم فيها كتابة إجراء يتم نداءه عن طريق برنامج آخر هو عميل لخدمة الويب web service client. مثلاً يقوم أحد المبرمجين بكتابة خدمة ويب لحجز تذكرة سفر لصالح شركة خطوط طيران مثلاً، وذلك بدلاً من أن يكون البرنامج في شكل صفحة عن طريق الإنترنت. فيقوم مبرمج آخر يعمل لصالح وكالة سفر بها برنامج لإدارة الوكالة أن يقوم بتضمين نداء إجراء حجز التذكرة من ذلك البرنامج بدلاً من أن يقوم الموظف بفتح صفحة في الإنترنت لحجز تلك التذكرة. وليس شرط أن يتم نداء الإجراء بنفس اللغة التي تمت كتابة خدمة الويب بها. وهذه ميزة مهمة في برامج خدمات الويب.

وهذا هو مثال لطريقة تصميم نظام به خدمة ويب ويُسمى نظام متعدد الطبقات:




نجد أنه في التصميم توجد ثلاث طبقات:

- قاعدة البيانات والتي تحتوي على بيانات المؤسسة
- مخدم برامج خدمات الويب ويُسمى أحياناً بالطبقة الوسيطة middle tier وهو يحتوي على إجراءات تحتاجها البرامج العميلة يتم تنفيذها في قاعدة البيانات أو أي مورد آخر من موارد المؤسسة
- الطبقة الأخيرة هي طبقة برنامج العميل client application وبه نداء لإجراءات يتم تنفيذها في المخدم الوسيط.

طريقة معمارية تعدد الطبقات لها عدة فوائد منها:

1. عزل قاعدة البيانات ومخدمها عن الأجهزة العميلة، وهذا يُقلل نقاط الإتصال على قاعدة البيانات. فإذا كانت مؤسسة بها مائة عميل مثلاً، فبدلاً من أن يتم عمل مائة نقطة إتصال مباشر مع قاعدة البيانات من أجهزة العملاء، يتم تجميع تلك الإتصالات في مخدم وسيط واحد أو إثنين وبدورها تقوم تلك الأجهزة الوسيطة بالتعامل مع قاعدة البيانات.

2. لانحتاج لتثبيت مكتبات للوصول لقاعدة البيانات في أجهزة العملاء، فقط يكفي تثبيت مكتبة التعامل مع قاعدة البيانات في الأجهزة الوسيطة. فإذا تم تغيير تلك المكتبة أو حتى إذا تم تغيير محرك قاعدة البيانات نفسها يتم عمل هذا التغيير في البرامج الوسيطة فقط.

3. زيادة تأمين وسرية قاعدة البيانات. حيث قمنا بعزل العميل عن قاعدة البيانات. فيُمكن أن يتم حصر سماحية الوصول إلى قاعدة البيانات عن الأجهزة الوسيطة فقط.

4. وسيلة إتصال ومخاطبة برمجية بين المؤسسات المختلفة. فلا يمكن لبنك مثلاً أن يقوم بالسماح لبنك آخر للدخول على قاعدة بياناته لتنفيذ عمليات معينة، إنما يتم عمل خدمات ويب محصورة في هذه الخدمات التي يطلبها البنك الآخر وإعطاءه صلاحية لندائها.

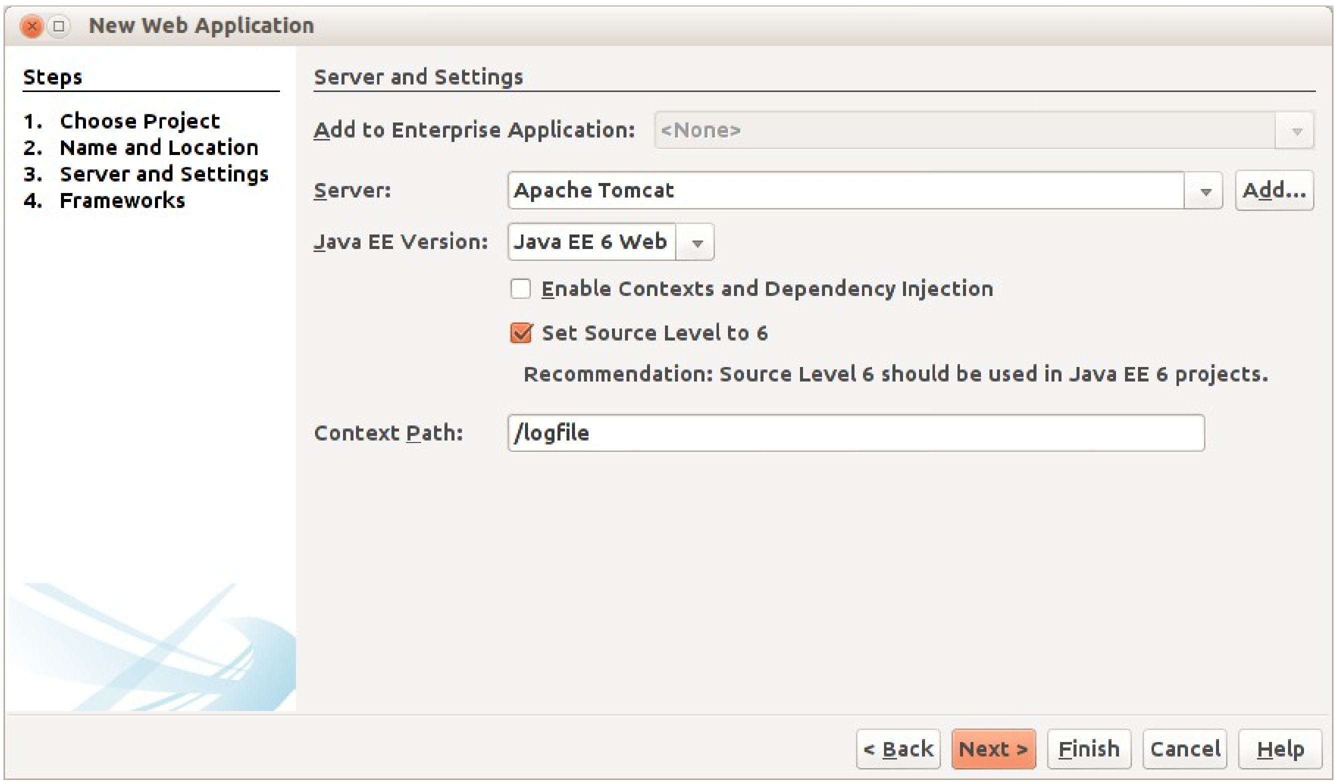
5. وسيلة إتصال بين الأنظمة المختلفة في المؤسسة الواحدة. حيث يُمكن لمؤسسة أن يكون لديها أكثر من نظام من جهات مختلفة، ولتكامل تلك الأنظمة مع بعضها يُمكن أن يوفر كل نظام خدمات ويب تسمح للأنظمة الأخرى الإستفادة منه. فمثلاً إذا كان هُناك نظام لإرسال رسائل نصية فبدلاً من إعطاء البرامج الأخرى صلاحية على قاعدة البيانات لإرسال تلك الرسائل يُفضل أن يكون لديه خدمات ويب لإرسال وإستقبال الرسائل الموجهة إلى البرامج الأخرى.

6. التقليل من التحديثات المستمرة في برامج العملاء. ففي أغلب الأحيان يكون التحديث والإضافات في النظام تحدث في قاعدة البيانات والطبقة الوسيطة ولاتتأثر البرامج الطرفية في أجهزة العملاء بهذا التغيير، فنقلل بذلك تكلفة صيانة ومتابعة النُسخ عند المستخدمين.

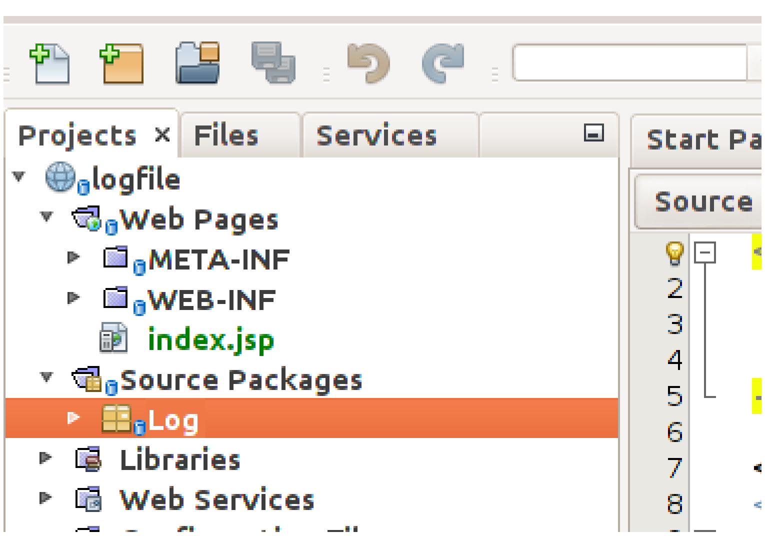
# برنامج خدمة ويب للكتابة في ملف

في هذا المثال نريد كتابة خدمة ويب من نوع تقنية الـ SOAP بها إجراء لإستقبال نص وكتابته في ملف نصي، ثم كتابة إجراء آخر لقراءة محتويات الملف النصي الذي تتم الكتابة فيه.

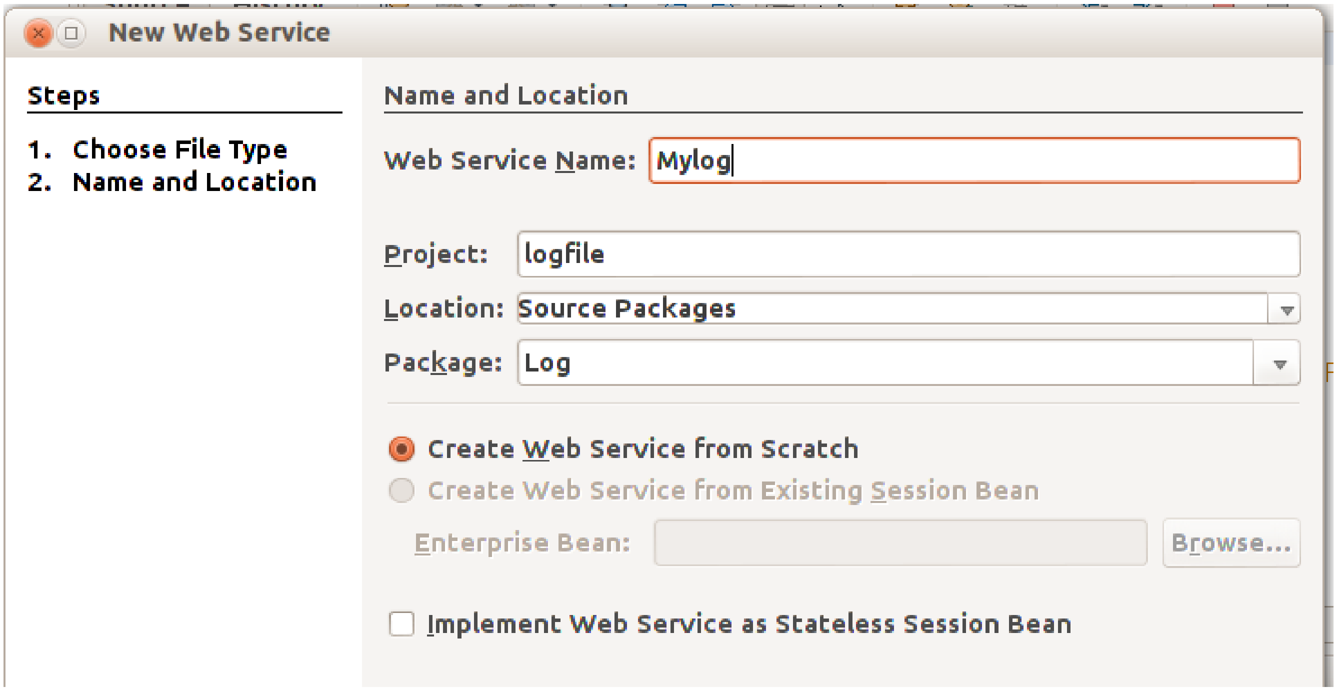
نقوم بإنشاء برنامج جديد من نوع Java Web/Web Application ونسميه *logfile* ثم نختار tomcat كمخدم ويب له:




بعد ذلك نقوم بإضافة حزمة جديدة نُسميها Log في Source Packages:



وفي الحزمة الجديدة Log نقوم بإضافة Web Service نسميها Mylog كما في الصورة التالية:




بعدها يتم التنبيه على أنه سوف يتم إضافة مكتبة METRO، فنقوم بإختيار موافقة.

يتم إضافة الكود التلقائي التالي في الملف Mylog.java:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package Log;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

/**
 *
 * @author motaz
 */
@WebService(serviceName = "Mylog")
public class Mylog {

    /**
     * This is a sample web service operation
     */
    @WebMethod(operationName = "hello")
    public String hello(@WebParam(name = "name") String txt) {
        return "Hello " + txt + " !";
```

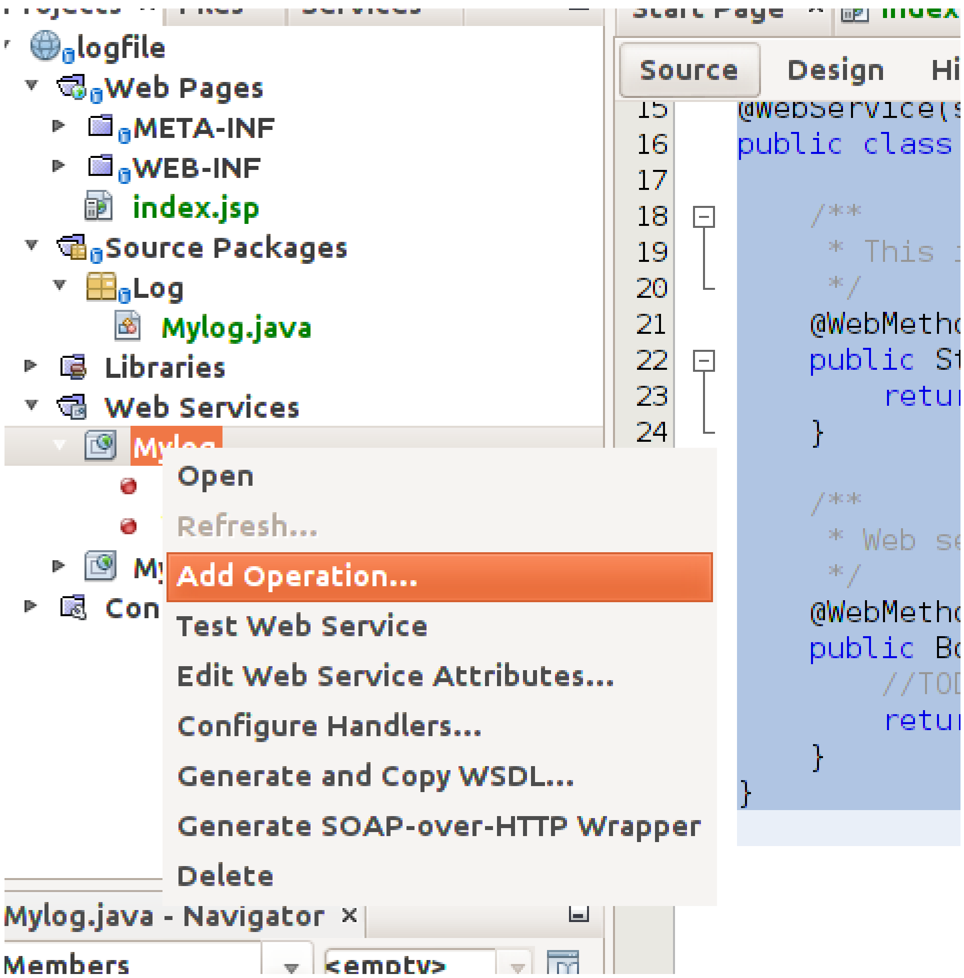
```
    }
}
```

أولاً نقوم بإضافة متغير مقعطي اسمه lastError في بداية تعريف خدمة الويب لنضع فيها الأخطاء التي تحدث:

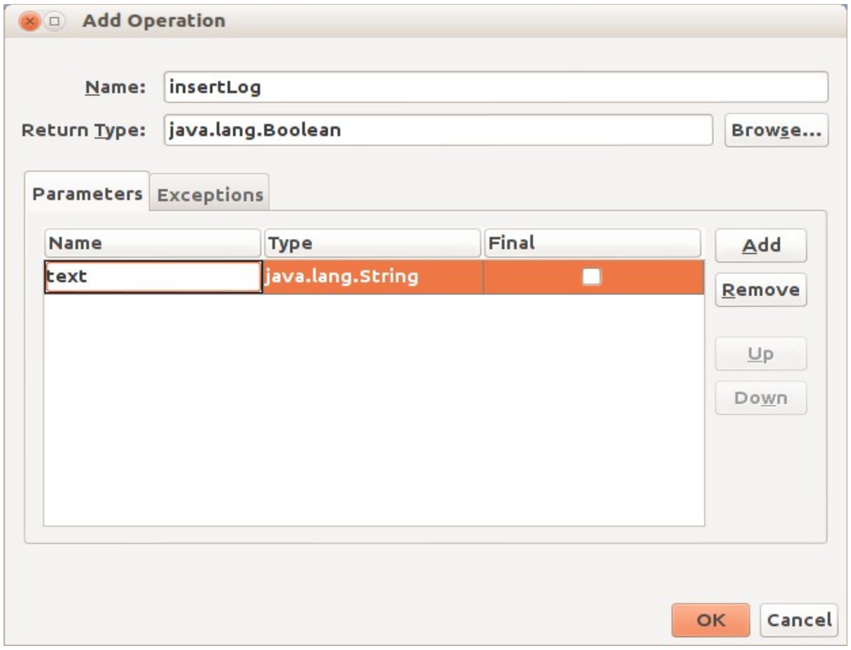
```
@WebService(serviceName = "Mylog")
public class Mylog {

    public String lastError = "";
```

ونجد أيضاً أنه تم إضافة فرع جديد في المشروع إسمه Web Services عند فتحها نجد Mylog، فنضيف إجراء جديد فيه بواسطة Add Operation



نقوم بتسمية ذلك الإجراء insertLog. وهو يرجع النوع boolean ويستقبل متغير اسمه text من النوع المقطعي String كما في الصورة التالية:




وإذا رجعنا مرة إخرى للملف Mylog.java نجد أنه تم إضافة الإجراء insertLog :

```
@WebService(serviceName = "Mylog")
public class Mylog {

    /**
     * This is a sample web service operation
     */
    @WebMethod(operationName = "hello")
    public String hello(@WebParam(name = "name") String txt) {
        return "Hello " + txt + " !";
    }

    /**
     * Web service operation
     */
    @WebMethod(operationName = "insertLog")
    public Boolean insertLog(@WebParam(name = "text") String text) {
        //TODO write your implementation code here:
        return null;
    }
}
```

بعدها قمنا بإستلاف إجراء الكتابة في ملف نصي من مثال سابق وعمل بعض التعديلات:

```java
    private boolean writeToTextFile(String aFileName, String text)
    {
     try{
       FileOutputStream fstream = new FileOutputStream(aFileName, true);

       DataOutputStream textWriter = new DataOutputStream(fstream);

       textWriter.writeBytes(text);
       textWriter.close();
       fstream.close();
       return (true); // success


      }
        catch (Exception e)
      {
        lastError = e.getMessage();
        return (false); // fail
      }

   }
```

وأضفناه في نهاية الملف Mylog.java ليتم إستدعاءه من الإجراء insertLog بالطريقة التالية:

```java
    @WebMethod(operationName = "insertLog")
    public Boolean insertLog(@WebParam(name = "text") String text) {

        boolean result = writeToTextFile("/tmp/mylog.txt", text);

        return result;
    }
```

ثم اضفنا إجراء آخر للقراءة أسميناه readLog بواسطة Add Operation كما في المثال السابق لكن بدون أن تكون له مدخلات، فقط مخرجات في شكل مقطع. فتتم إضافته بالشكل التالي:

```java
    @WebMethod(operationName = "readLog")
    public String readLog() {
        //TODO write your implementation code here:
        return null;
    }
```

ثم قمنا بكتابة إجراء القراءة من ملف نصي لإرجاع الملف كاملاً في متغير مقطعي بدلاً من عرضه على الشاشة:

```java
 private String readTextFile(String aFileName)
    {
     try{
       BufferedReader reader = new BufferedReader(new FileReader(aFileName));
       String contents = "";
       String line = reader.readLine();
```

```
     while (line != null) {
         contents = contents.concat(line + "\n");
         line = reader.readLine();

     }

     reader.close();

     return (contents);


    }
      catch (Exception e)
    {
      lastError = e.getMessage();
      return (null); // fail
    }

 }
```

قمنا بنداء القراءة من الملف النصي في الإجراء readLog بالشكل التالي:

```
  @WebMethod(operationName = "readLog")
  public String readLog() {

      String filetext = readTextFile("/tmp/mylog.txt");
      return filetext;
  }
```

وفي النهاية قمنا بكتابة إجراء لإرجاع آخر خطأ حدث وأسميناه getLastError:

```
  @WebMethod(operationName = "getLastError")
  public String getLastError() {
      //TODO write your implementation code here:
      return lastError;
  }
```

حيث يستخدمه العميل لمعرفة الخطأ الذي حدث في خدمة الويب أثناء ندائها.
نلاحظ أنه لابد أن نستخدم دليل به صلاحية للمستخدم tomcat6 أو tomcat7 والذي يتم إستخدامه مع نظام الشغيل عند التعامل مع خدمات الويب. وفي هذا المثال السابق استخدمنا الدليل tmp/ بإعتبار أن به صلاحية لكافة المستخدمين في بيئة لينكس.

في الواقع العملي تكون إجراءات خدمة الويب مرتبطة بتنفيذ إجراءات في قواعد بيانات مثلاً إدخال قيد محاسبي، إدراج معاملة بنكية، دفع فاتورة هاتف. كذلك يُمكن أن تقوم خدمات الويب بنداء خدمات ويب أخرى، فيصبح المعمارية ذات أربع طبقات: عميل - خدمة ويب - خدمة ويب أخرى - قاعدة بيانات.

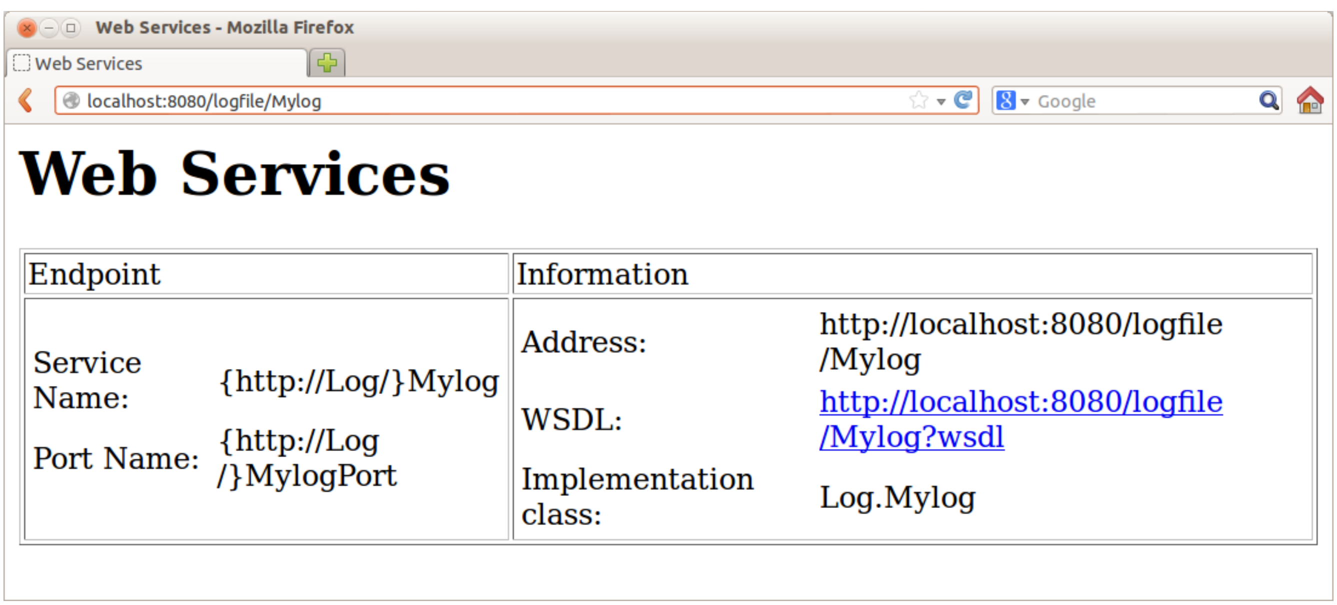
بعد ذلك يُمكن تشغيل البرنامج فيتم فتح متصفح الويب تلقائياً لتظهر الشاشة التالية:




بعد نهاية عنوان الويب نقوم بإضافة إسم خدمة الويب Mylog لِيُصبح العنوان هو:

http://localhost:8080/logfile/Mylog

فيظهر لنا معلومات خدمة الويب Mylog:



وعند الضغط على عنوان الـ WSDL يظهر لنا ملف XML وهو وصف لخدمة الويب ويُستخدم عند عمل البرامج العميلة لخدمات الويب:




بذا نكون قد إنتهينا من كتابة وتشغيل خدمة الويب في مخدم Tomcat.

وهذا هو رابط الـ WSDL:

http://localhost:8080/logfile/Mylog?wsdl

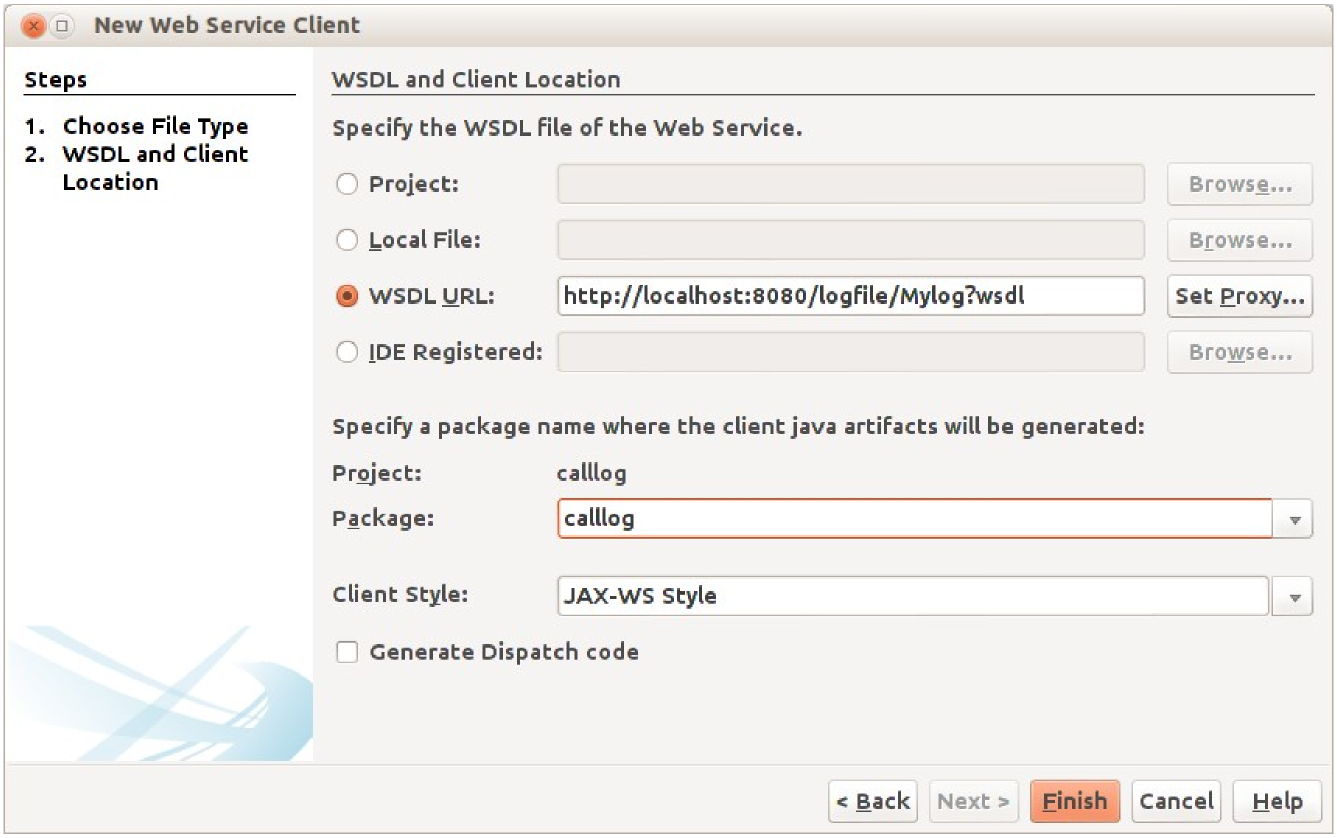
وهذا هو الشيء الوحيد الذي يحتاجه المبرمج لكتابة برنامج عميل لإستخدام خدمة الويب. ويمكن أن يقوم بإستخدام أي لغة برمجة تدعم تقنية الـ SOAP لنداء الدالتين insertLog و readLog.

# برنامج عميل خدمة ويب

يُمكن أن يكون إجراء نداء خدمة الويب في أي نوع من البرامج، مثلاً يُمكن أن يكون في برنامج سطح مكتب Desktop application أو برنامج ويب أو حتى برنامج سطر الأوامر كما في مثالنا التالي.

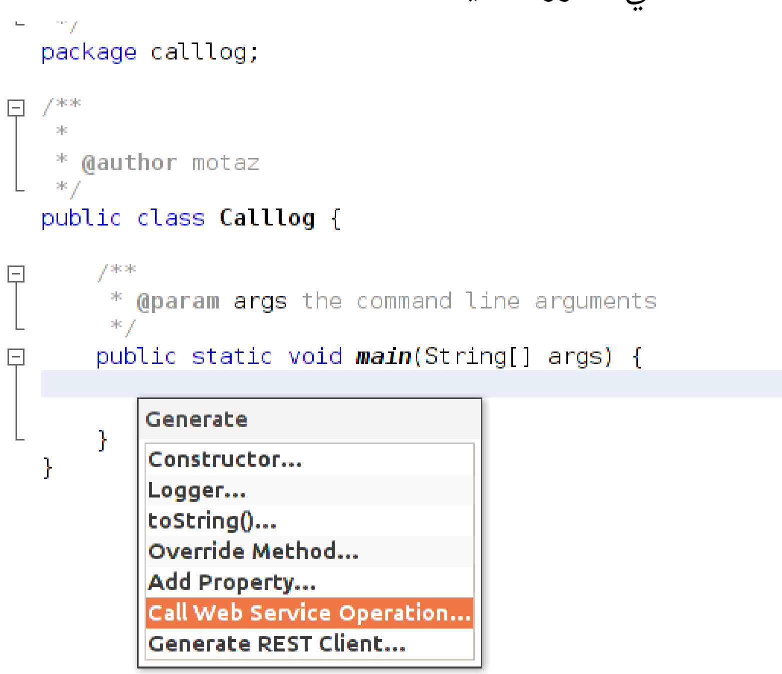
نقوم بإنشاء برنامج جديد من نوع Java/Java application نُسميه calllog.

بعدها نجد أن هُناك حزمة اسمها calllog في البرنامج. نقوم بإضافة عميل خدمة ويب بواسطة new Web service client فيظهر لنا الفورم التالي:

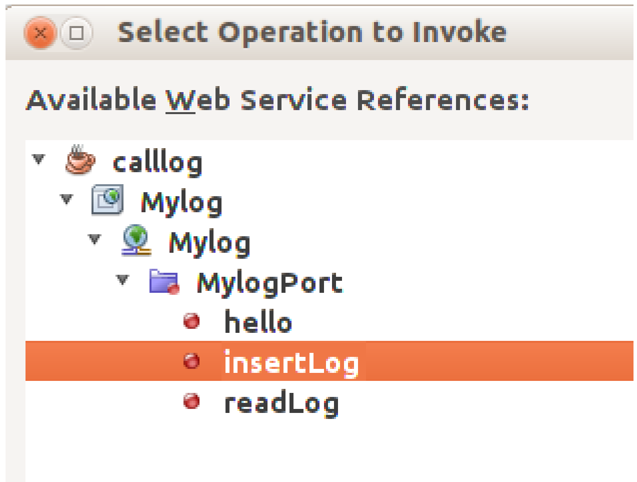



حيث نختار WSDL URL نضع فيه عنوان الـ WSDL لخدمة الويب. ثم نختار الحزمة calllog في Package ثم نضغط الزر Finish

نرجع لملف الكود الرئيسي calllog.java فنقوم بالضغط بالزر اليمين للماوس داخل الإجراء main ونختار call Web service operation كما في الصورة التالية:



ثم إختيار insertLog:




فيتم إضافة إجراء جديد لنداء خدمة الويب بالشكل التالي:

```
    private static Boolean insertLog(java.lang.String text) {
        calllog.Mylog_Service service = new calllog.Mylog_Service();
        calllog.Mylog port = service.getMylogPort();
        return port.insertLog(text);
    }
```

ونكرر نفس العملية السابقة لإضافة نداء الإجراء readLog:

```
    private static String readLog() {
        calllog.Mylog_Service service = new calllog.Mylog_Service();
        calllog.Mylog port = service.getMylogPort();
        return port.readLog();
    }
```

ثم قمنا بتعديلها لإضافة إظهار الخطأ الذي يحدث في خدمة الويب:

```
    private static Boolean insertLog(java.lang.String text) {
        calllog.Mylog_Service service = new calllog.Mylog_Service();
        calllog.Mylog port = service.getMylogPort();
        boolean res = port.insertLog(text);
        if (!res)
                System.out.println("Error: " + port.getLastError());
      return(res);
```

```
    }

    private static String readLog() {
        calllog.Mylog_Service service = new calllog.Mylog_Service();
        calllog.Mylog port = service.getMylogPort();
        String result = port.readLog();
        if (result == null) {
            System.out.println("Error: " + port.getLastError());
        }
        return (result);
    }
```

ثم قمنا بنداء الإجرائين في الدالة الرئيسية للبرنامج:

```
    public static void main(String[] args) {

        Date today = new Date();
        insertLog(today.toString() + ": Sample text\n");
        String result = readLog();
        System.out.print(result);

    }
```

عند تشغيل البرنامج نتحصل على الخرج التالي:

```
Fri Mar 29 12:34:48 EAT 2013: Sample text
Fri Mar 29 12:34:53 EAT 2013: Sample text
```

عند تنفيذ أي من الإجراءات في الجهاز العميل فإنه يتم تنفيذه في المخدم.

وفي الواقع تكون خدمة الويب في جهاز منفصل والبرنامج العميل يكون متصلاً به عبر شبكة محلية أو شبكة الإنترنت، وكل تعقيدات الإتصلات بقواعد البيانات يكون في جهة خدمة الويب، ويكون برنامج العميل مبسطاً بقدر الإمكان لتحقيق فوائد معمارية تعدد الطبقات.

وفي الختام نتمنى أن تُنال الفائدة من هذا الكتاب.

معتز عبدالعظيم الطاهر

كود لبرمجيات الكمبيوتر

[code.sd](code.sd)